Universidad Carlos III de Madrid

Escuela Politécnica Superior

Ingeniería Técnica en Informática de Gestión

Proyecto de Fin de Carrera

Análisis e Implementación de un jugador automático de Five Lines

Miércoles, 28 de octubre de 2015

Autor: José Tomás Nogales Nieves
Tutor: Carlos Linares López

"For young players, classic games are brand new. For older players, they bring back memories and make you feel good"

Satoru Iwata (1959 – 2015)

Agradecimientos

Quiero agradecer el apoyo recibido por todas esas personas cercanas sin las cuales, posiblemente, no hubiese llegado a donde estoy.

A mi familia, a la que no puedo hacer justicia con sólo unas líneas. A mi madre por ser tan insistente y saber escuchar, a mi padre por ser atento y aconsejar, y a mis hermanas por estar ahí para todo lo necesario, sin importar que cada uno estemos ahora en un sitio diferente.

A mis amigos que me esperan en Madrid: a Kike por servir de inspiración, a Edu por siempre ser fuente de diversión, a Mateo por siempre estar de buen humor, a Albert por ser el primer amigo que allí tuve y a Pablo por ser el último que allí hice.

También a la gente que ha hecho que mi estancia en Darmstadt sea mejor de lo que pensaba que sería. A Enric, Vicky, Andreas, Silvia, Giuseppe, Anton y Jean-Baptiste.

Y, para finalizar, agradecer a Álvaro Torralba de Reyna, por su ayuda, entusiasmo y por ser, además, un muy buen amigo; y a mi tutor en ésta última etapa de la carrera, Carlos Linares López, por todo lo que ha hecho para que éste proyecto salga adelante.

Índice

Índice de ilustraciones

Índice de Tablas

Capítulo 1
Introducción

Los videojuegos son casi tan antiguos como el concepto de la informática. El primer ejemplo considerado como tal data de 1947, cuando Thomas T. Goldsmith Jr y Estle Ray Mann solicitaron una patente para un "Dispositivo de entretenimiento de tubo de rayos catódicos", el cual, basado en tecnología de radar, consistía en controlar un punto en la pantalla simulando un misil siendo disparado contra varios objetivos, los cuales eran dibujos fijos en la pantalla. Desde entonces y hasta el día de hoy, los videojuegos han ido ganando en complejidad y presencia en la sociedad constantemente, siendo la forma de ocio preferida de millones de personas.

Unido a la proliferación de sistemas cada vez más complejos en los videojuegos, y siempre con el afán de mantener al jugador entretenido o sumergido en los mundos que éstos crean, está el campo de la inteligencia artificial. No todos los juegos requieren de ella, y las técnicas de inteligencia artificial que los juegos emplean son tan variadas como los juegos mismos.

El motivo que me ha llevado a escoger un tema relacionado con un videojuego, así como el decidir dedicarme a la informática, viene dictado en gran medida por la afición que he tenido por éste sector desde mi infancia. Desde que puedo recordar, siempre he tenido a mi disposición al menos un aparato, fuese ordenador o videoconsola, en el que poder disfrutar de ellos. Incluso en el día de hoy, pese a no disponer del tiempo suficiente para jugar todo lo que me gustaría, siempre intento sacar tiempo de vez en cuando para poder dedicarme a ellos o, al menos, estar informado de los cambios que éstos sufren.

El juego que nos ocupa fue sugerido por mi tutor del proyecto de final de carrera, y pronto vi que se trataba, por puntos que explicaremos a continuación, de un caso muy interesante y, por tanto, tardé poco en decidirme a abordarlo.

La presente memoria cuenta con las siguientes secciones:

- **Capítulo 1, Introducción:** La sección actual

- **Capítulo 2, Estado de la cuestión:** Donde hablaremos del juego original y algunas de las variaciones que éste ha sufrido a lo largo de los años. También comentaremos algunos de los algoritmos que se han considerado para abordar el proyecto.
- **Capítulo 3, Objetivos:** Donde detallaremos lo que pretendemos conseguir con éste proyecto.
- **Capítulo 4, Desarrollo:** Donde hablaremos en profundidad de las distintas por las que hemos pasado hasta llegar a la aplicación final.
- **Capítulo 5, Resultados:** Donde expondremos los resultados obtenidos por nuestros agentes, comparándolos y comentándolos entre sí.
- **Capítulo 6, Líneas Futuras:** Donde sugeriremos que acciones podrían tomarse con la aplicación final con el motivo de dar ideas para ampliarla.
- **Capítulo 7, Conclusiones:** Donde estudiaremos si hemos conseguido alcanzar los objetivos que nos hemos propuesto
- **Capítulo 8, Planificación y presupuesto:** Donde veremos la planificación y presupuestos originales y las contrastaremos con el desarrollo y costes reales.

Capítulo 2
Estado de la cuestión

Esta sección describe el origen del juego que va a ser el foco del proyecto, *Five Lines* (*Color Lines* en su versión original), discutiendo la historia del original y detallando sus reglas y posibles variaciones. Tras hablar del juego, se procederá a detallar algunos algoritmos de Inteligencia Artificial, comentando por qué se ha decidido o no utilizar dicho algoritmo para este proyecto.

2.1. Introducción al Five Lines

A continuación hablaremos un poco del juego del que trata el proyecto, comentando sus orígenes y discutiendo el juego de reglas común que todas las versiones contienen. Posteriormente hablaremos un poco de algunas de las variaciones existentes sobre el juego original y, finalmente, sobre la versión implementada para este proyecto.

2.1.1. Historia y versiones

Color Lines, también conocido como *Lines*, es un videojuego de puzle para un solo jugador inventado por Oleg Demin e introducido al mundo en 1992 por la compañía rusa Gamos (Геймос). Fue distribuido originalmente para MS-DOS.

En 1995 apareció la primera versión del juego para Windows (Windows 95), desarrollada por Igor Nedelko y Andrey Akselrod y distribuida por la compañía AbrewSoft como shareware. Esta versión fue un remake fiel al original y contaba con varias mejoras: gráficos de 256 colores, mejoras en la interfaz… También introdujeron nuevos elementos al juego como bolas de cuatro colores para niveles de dificultad más avanzados.

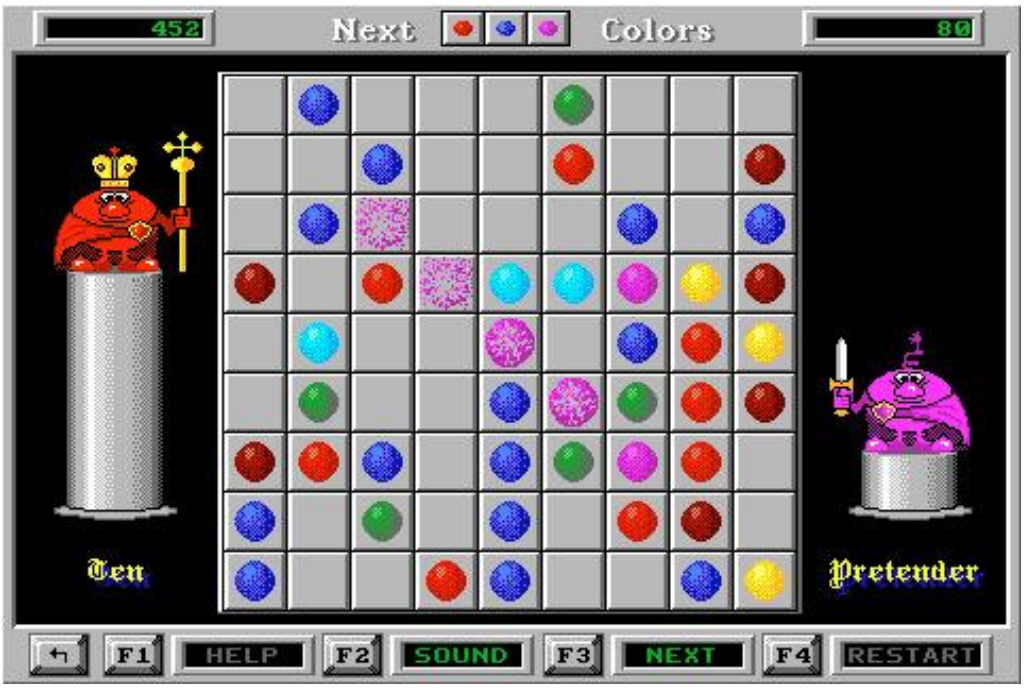

Ilustración 1 – Color Lines

Posteriormente, nuevas versiones del juego han ido apareciendo con el paso de los años. Algunas de las más destacables son las siguientes:

- *Five or more* (en español *Cinco o más*): esta versión está desarrollada para GNU+Linux y disponible por defecto en algunas distribuciones, como Ubuntu. Dispone de tres tableros distintos, que afectan a la dificultad.
- *Rainbowlines*, disponible on-line y para algunos dispositivos móviles. Dispone de numerosos cambios sobre el juego original que también serán comentados más adelante.
- *Park-o-Nora*, versión para iOS. La principal diferencia con el juego original es que éste trata sobre una oveja aparcando coches para ganar dinero. Al estar pensada para dispositivos móviles, el tablero es más pequeño y sólo se necesita hacer líneas de longitud cuatro.

2.1.2. Reglas y elementos del juego

En el juego *Color Lines* los elementos que definen el estado de la partida son, por defecto, los siguientes:

- Un tablero de 9x9 donde las bolas irán apareciendo.
- Bolas de siete colores posibles.
- Tres bolas de entre esos siete colores que aparecerán en el tablero la próxima vez que sea posible. Éstas bolas son conocidas por el jugador, pero no las posiciones que ocuparán cuando finalmente entren en juego.
- Longitud de línea mínima, cinco.
- Puntuación actual que el jugador ha acumulado.

El juego original sigue el siguiente esquema:

- La partida comienza con un tablero de 9x9 el cual contiene tres bolas en casillas al azar y elegidas entre siete posibles colores.
- El jugador puede, cada turno, seleccionar una de las bolas que hay en el tablero y moverla a cualquier otra posición vacía que desee, siempre y cuando haya un camino válido. Se entiende por "camino válido" una sucesión de casillas vacías unidas entre sí horizontal y/o verticalmente, sin importar la distancia ni el número de casillas que haya que recorrer.
- Si tras el movimiento del jugador hay conectadas al menos cinco (5) bolas del mismo color en una línea ("línea" a partir de ahora), sea ésta horizontal, vertical o diagonal; las bolas conectadas de esta forma desaparecerán, se aumentará la puntuación del jugador en el incremento correspondiente y el jugador podrá realizar un movimiento adicional usando las bolas presentes en el tablero.
- Si tras el movimiento del jugador no hay una línea, aparecerán en posiciones vacías al azar tres nuevas bolas en el tablero.
- El juego se repite hasta que el tablero está lleno y el jugador no puede realizar más movimientos.

Dadas estas condiciones, hay dos objetivos principales que el jugador intentará alcanzar:

- Conseguir la máxima puntuación posible.
- Hacer el mayor número de líneas posible en una partida, aumentando el número de turnos hasta que el juego finalmente acabe.

Aunque puedan parecer iguales, estos dos objetivos pueden entrar en conflicto en según la función empleada para calcular la puntuación. Distintas versiones usan distintas puntuaciones. Por ejemplo, en el caso de *Five or more*, tenemos la siguiente tabla:

Longitud de línea	Puntuación obtenida
5	10
6	12
7	18
8	28
9	42
10	82
11	108
12	138
13	172
14	210

Tabla 1 – Puntuación por línea en *Five or more*

2.1.3 Variaciones existentes sobre el juego

A lo largo de los años han ido apareciendo numerosas versiones sobre *Color Lines* para todo tipo de plataformas. Algunas son réplicas exactas del juego original, otras introducen cambios menores que no afectan mucho al juego en sí, como pueden ser niveles de dificultad, cambios

en el sistema de puntuación... Otras, sin embargo, introducen o cambian suficientes elementos como para que puedan considerarse un juego diferente.

A continuación se van a estudiar dos variaciones, una que es bastante fiel al original (*Five or more*) y otra que no lo es.

2.1.3.1 Five or more

Five or more, en su versión para GNU+Linux es en esencia el *Color Lines* original.

Ilustración 2 – *Five or more*, tablero grande

Sin embargo, además de la partida con los parámetros del original, ofrece la posibilidad de elegir entre tres tamaños de tablero, modificando algunas de las variables. Las posibilidades son:

- Pequeño:
  - Tablero de 7x7.
  - Cinco colores.
  - Tres bolas nuevas cada turno.
- Normal: Mismos parámetros que el juego original.
- Grande:
  - Tablero de 20x15.
  - Siete colores.
  - Siete bolas nuevas por turno.

Independientemente del modo seleccionado, las líneas deberán tener una longitud mínima de cinco.

2.1.3.2 Rainbowlines

De los juegos estudiados, éste es el que más opciones diferentes ofrece al jugador. Antes de comenzar la partida, el jugador tiene que seleccionar tres opciones, jugando con la combinación resultante.

Ilustración 3 - Rainbowlines

En primer lugar se selecciona el modo de juego. Éste modo cambiará de una forma u otra las reglas del juego y, por tanto, cómo debe el jugador enfrentarse a él.

- *Classic* (Clásico): El mismo modo que el *Color Lines* tradicional, el juego va por turnos, es decir, las bolas no aparecerán hasta que el jugador realice un movimiento que no resulte en línea. Durará hasta que el jugador no pueda ser capaz de realizar más movimientos.
- *Action* (Acción): Las bolas aparecen en el tablero tras pasar un tiempo, independientemente del jugador. En éste modo el jugador no dispone de tanto tiempo para pensar en el movimiento, pero por otra parte es capaz de realizar tantos movimientos como quiera sin que las bolas aparezcan. Hacer líneas sólo afecta a la puntuación. Teóricamente sería posible jugar hasta el infinito en éste modo, si el jugador dispone de una capacidad de análisis y reflejos lo suficientemente elevados.
- *Time attack* (Contrarreloj): similar al modo clásico con la excepción de que el jugador ahora está sujeto a un tiempo límite para jugar. Hacer una línea incrementa un poco el tiempo que el jugador dispone para jugar, además de retrasar la aparición de nuevas bolas por un turno.

Una vez elegido el modo del juego, el jugador deberá seleccionar un “color” entre siete. Esto afectará a la forma del tablero en el que jugará. Todos ellos son, de base, tableros 9x9. Sin embargo, salvo uno, todos tienen huecos en los que ni el jugador ni el juego podrán colocar

bolas en ellas. Esto afecta considerablemente a la dificultad, ya que los huecos y las formas que éstos le dan al tablero limitan las líneas que se pueden hacer en él.

Finalmente, el jugador tendrá que elegir entre dos dificultades, lo que afecta a la longitud mínima de la línea. En el modo de dificultad fácil la longitud mínima es de cuatro y en el difícil es cinco.

Un último detalle a destacar de esta implementación del juego es el cómo gestiona la aparición de nuevas bolas en el tablero. Aunque las posiciones y los colores se determinan al azar, el juego informa al jugador dónde pretende colocarlas. Si el jugador decide colocar una bola donde otra va a aparecer, el juego escoge otra casilla vacía para esa bola, pero manteniendo las otras dos. Esto permite al jugador planificar considerablemente mejor sus próximos movimientos, restándole mucha dificultad al juego en comparación con el *Color Lines*.

2.1.4 Five Lines

Five Lines, como nos referiremos al juego a partir de ahora, es la implementación del juego desarrollada para este proyecto. Tiene las siguientes características:

Parámetros de juego configurables: antes de iniciar la partida, todos los parámetros se pueden configurar como se desee:

- Tablero *MxN*.
- Número de bolas a aparecer el próximo turno.
- Número de tipos o colores, sin existir tipos mixtos (i.e.: bolas de más de un color o comodines).
- Longitud mínima de línea.

Los valores por defecto, y los empleados para los estudios con las diferentes IAs son los del *Color Lines* original.

Al contrario que el resto de las implementaciones, lo que el juego emplea como puntuación es el número total de líneas realizadas, independientemente de las bolas eliminadas por línea. Se decidió en un principio que las IAs jugarían a intentar sobrevivir la mayor cantidad de turnos posibles, priorizando el hacer líneas lo antes posible, y por tanto no se ha implementado un sistema de puntuación propiamente dicho.

2.2. Algoritmos de búsqueda en Inteligencia Artificial

En esta sección describiremos algunos de los algoritmos considerados para implementar un jugador inteligente para el juego *Five Lines*. Pero primero, hablaremos de los dos algoritmos de búsqueda no informada más básicos: búsqueda en amplitud, búsqueda en profundidad.

2.2.1 Búsqueda en amplitud

La búsqueda en amplitud se basa en ir explorando, progresivamente, todos los estados o nodos posibles que nacen del estado actual, e ir bajando en el árbol de exploración un nivel cada vez, cuando el nivel actual se ha explorado por completo.

En esencia, dado un nodo raíz (nivel 0), se generan todos los nodos hijos posibles y se exploran. Explorarlos significa:

1. Se comprueba si el nodo hijo (nivel 1) es la "solución" al problema. Si lo es, la búsqueda termina. Si no lo es se continúa, se generan los nodos hijos (nivel 2) de éste nodo a explorar y se pasa al siguiente nodo (nivel 1), hijo del raíz.
2. Se repite el paso anterior para el resto de nodos hijos del nodo raíz hasta que se han explorado todos y han sido expandidos.
3. Si no se ha encontrado solución, se procede desde el paso 1 al nivel siguiente (nivel 2).
4. Se continúa hasta encontrar una solución.

Así, no se procederá hasta el nivel N+1 hasta que todos los nodos del nivel N han sido explorados. Este proceso nos asegura de que una vez se encuentre una solución, ésta será óptima, al ser la menos alejada del nodo raíz.

Ventajas:

- Asegura que la solución encontrada, de haberla, es la mejor o más óptima posible.

Desventajas:

- Es muy costoso de ejecutar ya que requiere almacenar en memoria todos los nodos que aún no han sido explorados. Dependiendo del problema, esto puede crecer exponencialmente, haciendo impracticable emplear éste algoritmo en gran cantidad de casos.

Conclusiones:

En el caso del *Five Lines*, esta aproximación a resolver el problema no nos es de mucha utilidad porque disponemos de un factor de ramificación muy elevado. Un solo nodo (movimiento ejecutado) tendría como hijos todas las posibles combinaciones de bolas en casillas desocupadas tras ejecutar el movimiento.

2.2.2 Búsqueda en profundidad

El algoritmo de búsqueda en profundidad se basa en explorar un nodo hasta agotar todas las posibilidades del mismo. Es decir, dado un nodo raíz (nivel 0):

1. Generamos los descendientes del nodo (nivel 1). Escogemos uno de esos descendientes (generalmente el primero) y comprobamos si es la solución que buscamos. Si no lo es procedemos.

2. Generamos los descendientes del nodo en el que nos hallamos (nivel 2), y repetimos. Éste proceso continuará hasta encontrar una solución o, de estar implementado, alcanzado el nivel límite N.
3. Si se ha alcanzado el límite N pero no se ha llegado a una solución, se retrocede a un nivel anterior y se explora en profundidad a partir de él.

Por lo tanto, en éste caso, para un nodo dado iremos descendiendo niveles hasta encontrar una solución a nuestro problema o alcanzar el límite de profundidad establecido (de haberlo). Si no se encuentra se subirá en el árbol los niveles necesarios.

Ventajas:

- Es potencialmente más rápido para encontrar una solución que la búsqueda en amplitud.
- Requiere una cantidad de recursos significativamente menor, al no necesitar tanta memoria para explorar nodos no visitados.

Desventajas:

- Si no hay establecido un límite de niveles y el problema lo permite, puede que nunca se alcance una solución. Incluso si hay límite, pero éste es demasiado grande, puede requerir más tiempo en encontrar una solución del que sería deseable.
- Si se encuentra una solución nada nos asegura que la solución alcanzada sea la mejor de las soluciones posibles, al contrario de lo que pasa con el algoritmo de búsqueda en amplitud.

Conclusiones:

Éste algoritmo tampoco es idóneo dado el alto número de descendientes posibles para un estado dado. Incluso estableciendo un límite bajo, por ejemplo tres niveles, la búsqueda tardaría demasiado tiempo debido al factor de ramificación tan elevado que nuestro problema tiene.

2.2.3 Minimax

El minimax es un algoritmo de decisión enfocado en minimizar la máxima pérdida en juegos o sistemas con un adversario. En resumen puede definirse en escoger la mejor jugada para uno (J1), suponiendo que el contrincante escogerá la mejor para sí. Esto sistema es particularmente adecuado para juegos de suma cero, donde las ganancias de un jugador implican pérdidas para los demás.

El algoritmo, en líneas generales consta de los siguientes pasos, para un nodo raíz en el turno del jugador para el cual se tiene que tomar la decisión:

1. Se genera el árbol de juego, todos los nodos hasta llegar al estado. Cada nivel del árbol representa a un jugador distinto. Suponiendo dos jugadores, los niveles impares serán

los del jugador (MAX) para el cual se está tomando la decisión y los pares los del oponente (MIN).
2. En los nodos terminales, se determina el estado de la partida o la ganancia/pérdida de ese nodo.
3. Se propaga el valor de los nodos terminales a los nodos superiores teniendo en cuenta qué jugador escogerá cada turno. Si es un nivel del jugador actual, un nodo padre tendrá el valor más alto de entre sus hijos (MAX) ya que estamos intentando maximizar ganancias. Si es un nivel del oponente, el nodo tendrá el valor más bajo de entre sus hijos (MIN), ya que se asume hará la jugada que más nos perjudica.
4. Se escoge la jugada con los valores que hayan llegado al nivel superior.

Este algoritmo es inviable para la mayoría de casos, exceptuando los más sencillos. Realizar una búsqueda completa conlleva un gran consumo de tiempo y memoria. Para hacerlo más manejable, se pueden aplicar restricciones al tiempo de ejecución o a la cantidad de niveles a explorar. Otra alternativa sería aplicar una poda alfa-beta, la cual asume que el oponente no permitirá que realicemos nuestras mejores jugadas.

Conclusiones:

En el caso del *Five Lines* se descartó utilizar este algoritmo principalmente porque el oponente del juego es sólo el azar. Para hacer viable un agente que emplease éste algoritmo, y teniendo en cuenta sobre todo el alto factor de ramificación del juego, deberíamos tomar una aproximación muy pesimista al juego (bolas cayendo siempre en los peores sitios, no recibiendo los colores apropiados a tiempo). Ésta no es una representación realista de un sistema donde el azar es el centro de todo y, por lo tanto, decidir movimientos de una manera siempre pesimista no parece la mejor opción. Especialmente porque el oponente del juego es indiferente a los resultados del jugador.

2.2.4 Montecarlo UCT

El algoritmo de Montecarlo es un método que se puede utilizar para tomar decisiones en sistemas muy complejos: sistemas con un espacio de estados muy elevado, sistemas con un factor de ramificación muy alto y/o sistemas donde hay incertidumbre:

1. Selección: comenzando por la raíz, se seleccionan recursivamente un nodo hijo óptimo hasta, llegar a un nodo hoja.
2. Expansión: Si el nodo seleccionado no termina la partida, se crean uno o más hijos de éste.
3. Simulación: A partir del nodo generado en el punto anterior, se ejecuta una simulación de la partida hasta llegar a un resultado.
4. Retro-propagación: Se actualiza el árbol de estados con los resultados obtenidos con la simulación.

Este proceso se repetirá hasta agotar el límite, sea éste tiempo o número de ejecuciones. Es importante que por cada nodo se guarden dos datos: el valor estimado (calculado a partir de los resultados de las simulaciones) y el número de veces que ese nodo se ha visitado.

Un punto muy decisivo en el algoritmo es la selección. Es importante intentar encontrar un equilibrio entre la exploración de nodos nuevos y la explotación de los mejores nodos para intentar encontrar la mejor solución posible.

Uno de los métodos de selección más empleados habitualmente, es la selección con UCB (*Upper Confidence Bounds*), que utiliza la siguiente fórmula:

$$v_i + C \times \sqrt{\frac{\ln N}{n_i}}$$

Ilustración 4 - Fórmula UCB

Dónde:

- v_i es el valor estimado del nodo.
- n_i es el número de veces que el nodo ha sido explorado.
- N es el número total de veces que su padre ha sido explorado.
- C es un parámetro de sesgo ajustable.

La fórmula UCB equilibra la *explotación* de nodos conocidos que ofrecen buenos resultados con la *exploración* de nodos relativamente poco visitados. Las estimaciones de los resultados que cada nodo ofrece están basadas en simulaciones aleatorias, por lo que ha de visitarse los nodos un cierto número de veces antes de que estas estimaciones sean de confianza. Las estimaciones serán poco fiables al comienzo de la búsqueda pero convergerán a estimaciones más fiables con el tiempo suficiente y a estimaciones perfectas si se dispusiese de tiempo infinito.

Ventajas:

- El algoritmo no requiere conocimiento del dominio para tomar decisiones razonables, suponiendo que dispone de tiempo suficiente. Puede funcionar efectivamente sólo conociendo los movimientos legales y las reglas del juego, por lo que una implementación puede reutilizarse para juegos diferentes con modificaciones mínimas.
- El árbol de búsqueda es asimétrico, ya que el algoritmo tiende a visitar los nodos más prometedores una mayor cantidad de veces. Esto hace que el algoritmo sea adecuado para juegos con factores de ramificación muy elevados como pueden ser el *Go* o el caso que nos ocupa, el *Five Lines*.

Inconvenientes:

- Es lento y puede tener problemas en encontrar una solución buena. Converger a una solución apropiada puede requerir un número muy elevado de iteraciones. Si el número de iteraciones o el tiempo de ejecución son lo suficientemente bajos, puede

tener problemas en encontrar una solución aceptable por no haber explorado nodos claves.

Conclusiones:

Éste algoritmo parece muy adecuado para el caso que nos ocupa ya que el juego tiene un factor de ramificación muy elevado y mucha incertidumbre. Los inconvenientes de éste algoritmo, no obstante son considerables. Por lo tanto, se aplicará una poda inicial al número de nodos a explorar, seleccionando los inmediatamente N mejores.

2.2.5 Round-Robin

Más que un algoritmo de búsqueda, Round Robin es un algoritmo de planificación. La idea principal de Round Robin es, dado un número de elementos en un grupo de trabajo, seleccionar cada uno de ellos de forma equitativa y siguiendo un orden racional. En su forma más básica, se ponen los elementos en una lista y se ejecutan desde el primero hasta el último. Una vez terminado, se repite el proceso hasta que se termine el trabajo.

Al contrario que el algoritmo en el punto anterior, Round Robin asegura que todos los nodos se van a explorar un mismo número de veces y por esto se ha considerado interesante. En el caso del *Five Lines*, dada la gran cantidad de posibles movimientos a explorar, si vamos a utilizar éste algoritmo se hará empleando un filtro previo a los mejores *N* movimientos.

2.3. Conclusiones

Tras evaluar un poco el juego y algunos algoritmos de búsqueda, podemos comentar lo siguiente:

1. El *Five Lines* es un juego para un solo jugador razonablemente popular. Prueba de ello son la cantidad de diferentes versiones que han aparecido y siguen apareciendo a lo largo de los años, el hecho de que esté incluido en diversos sistemas operativos de base y lo sencillo que resulta poder encontrarlo online.
2. Es un problema interesante de abordar desde el punto de vista de la Inteligencia Artificial debido a la gran cantidad de estados posibles que se pueden dar en una partida determinada.
3. Al ser el azar un factor tan importante, no habrá dos partidas iguales. Este azar determinará en gran medida la dificultad de una partida en concreto, y los resultados que se podrán obtener al final de ésta.
4. Es necesario jugar pensando un poco en los movimientos a realizar. Pese a ser imposible saber dónde se van a colocar las nuevas bolas tras ejecutar un movimiento, hay formas de minimizar el efecto que éste azar puede tener. Por lo tanto, hay estrategias más viables que otras.

Capítulo 3
Objetivos

El objetivo final de este proyecto es determinar cuál de entre los algoritmos o estrategias implementados es el que mejor funciona en el caso del *Five Lines*, e intentar discernir por qué.

3.1 Motivaciones del proyecto

Desde que los primeros ordenadores personales aparecieron, los videojuegos han estado presentes y han permitido la aparición de sistemas, reglas y mecánicas difíciles o imposibles de reproducir en medios tradicionales, y ello conlleva a nuevos problemas que explorar con IA.

Las razones principales por las cuales se ha optado por éste juego frente a otros posibles son las siguientes:

- Es un juego que ha sido razonablemente popular desde su concepción, como demuestran la cantidad de *remakes*, versiones y variaciones que han surgido y surgen a lo largo de los años. Una simple búsqueda en google mostrará numerosas páginas donde puede jugarse on-line gratuitamente.
- Es un problema interesante en búsqueda debido a los siguientes factores:
    - Es un puzle donde la capacidad de razonamiento es un factor importante. El modo de jugar y el optar por un movimiento sobre los demás impacta considerablemente en el resultado.
    - El azar es un elemento muy decisivo. Movimientos que se consideran "óptimos" o, al menos, razonablemente buenos para un estado del tablero pueden ser truncados por la aparición de bolas en lugares inapropiados, por la carencia de bolas de un color concreto en turnos siguientes o por ambos. Debido a la importancia del azar, pese a que es relativamente obvio seleccionar un puñado de movimientos que son mejores que otros, no es tan fácil decidir qué movimiento es, inherentemente, el mejor de todos.

  - Tiene un factor de ramificación muy elevado debido a la cantidad de movimientos posibles que hay en algunos momentos del juego y debido también a la gran aleatoriedad que el juego tiene. En el turno actual el jugador sólo conoce qué colores aparecerán en el tablero, lo cual ayuda a tomar decisiones, pero ignora dónde aparecerá cada uno, lo cual crea incertidumbre.
  - Con los parámetros del *Color Lines* original, es imposible no alcanzar el final de la partida. El objetivo por tanto no es "ganar" el juego, sino obtener la mayor puntuación posible. Alternativamente, se puede jugar a aguantar la mayor cantidad de turnos posibles (que es lo que se ha decidido que hará nuestro jugador).
- A día de hoy, nadie ha intentado desarrollar un agente para este juego.
- A título personal, el desarrollo de videojuegos y las posibles aplicaciones que la IA tiene dentro de él es un tema que me resulta muy interesante.

3.2 Objetivos

El objetivo principal de éste Proyecto de Fin de Carrera (PFC en adelante) es estudiar cómo pueden aplicarse las técnicas de búsqueda con incertidumbre en problemas conocidos, concretamente aplicándolo a un juego de ordenador.

En éste PFC se desarrollará un agente que, mediante técnicas de Inteligencia Artificial, sea capaz de jugar de un modo completamente autónomo y mostrando un comportamiento coherente al juego *Five Lines*, que es, como ya mencionamos en la sección 2.1.4, una versión del *Color Lines* desarrollada íntegramente para este proyecto con el fin de satisfacer todos los requerimientos que se han impuesto sobre los agentes. Se decidió por esta opción porque es menos costosa que encontrar una implementación ya hecha y acoplar los algoritmos de búsqueda a ella. Además, tratándose de un juego de un solo jugador, las posibilidades de que exista una implementación ya hecha a la que se le pueda añadir funcionalidad es remota.

Las variables del juego consideradas para éste PFC, en término de estudio de resultados, son las del *Color Lines* original:

- Tablero de 9x9 casillas.
- Bolas de siete posibles colores.
- Líneas de longitud cinco (con bolas del mismo color).
- Aparición aleatoria de tres bolas cada turno, conociendo los colores de cada una.

El objetivo principal del proyecto será diseñar un agente capaz de jugar de un modo inteligente en las variables mencionadas anteriormente. Sin embargo, dadas las características del juego y lo poco costoso que resulta cambiar los parámetros de la partida, se procurará emplear una algoritmia que permita que el agente juegue cambiando una o más de las anteriores variables: aparición de más o menos colores, número de bolas por turno, distinta longitud de línea o distinta dimensión del tablero ya sea en filas, columnas o ambas. Se mantendrán no obstante las siguientes restricciones:

- El tablero será siempre una matriz rectangular *MxN*, y éste será completo. En otras palabras, no se considerarán otras formas ni huecos en el tablero.
- Siempre se conocerá el número (suponiendo que sea aleatorio) y color de las bolas que aparecerán al finalizar el turno.
- La longitud mínima de línea es conocida al comenzar la partida.
- El número de colores es conocido al comenzar la partida.

Otro objetivo, en cierto modo secundario pero muy ligado al intentar desarrollar el mejor agente posible, será la implementación de varios agentes que emplearán algoritmos de búsqueda y decisión diferentes, con el objetivo de comparar los resultados obtenidos entre ellos y decidir así cual es la mejor estrategia. Los resultados se medirán en el número de líneas realizadas por partida, con lo cual se premiará el realizar las líneas lo antes posible, sin tener en cuenta la puntuación final. En el caso de alguno de los agentes se realizarán varios experimentos en los que se cambiarán algunos de sus parámetros. Estos parámetros se explicarán más adelante, en la sección de Resultados. Los agentes a desarrollar son los siguientes:

- *Random*: éste agente simplemente escogerá un movimiento al azar de entre todos los posibles. Aunque de antemano se prevé que ésta no es una estrategia válida, la función de éste agente es demostrar que no es posible obtener buenos resultados jugando al azar, y utilizar este agente como *baseline*.
- *Greedy*: éste agente escogerá siempre el movimiento que la función de evaluación implementada le dicte que es el mejor. El objetivo principal de éste agente es poder evaluar como de viable es la estrategia de considerar sólo el estado actual de la partida sin considerar turnos futuros (especialmente dónde aparecerán las bolas en el próximo turno, de qué colores y donde aparecerán las bolas en turnos venideros). También nos permite tener una estrategia base, razonablemente viable, contra la cual contrastar agentes que empleen algoritmos de búsqueda.
- *Montecarlo UCT*: éste agente empleará el algoritmo de búsqueda de Montecarlo, el cual, como se ha comentado en la sección 2, es un algoritmo bastante idóneo para problemas donde el azar es un elemento importante. Sin embargo, dado el factor de ramificación tan elevado, y considerando que, pasados sólo un par de turnos, las situaciones del tablero pueden ser tan elevadas, se ha optado por aplicar un filtro (utilizando la función de evaluación) para reducir los movimientos a estudiar a los *X* mejores y por no explorar nodos más allá de los *Y* más inmediatos. Tras estos *X* movimientos, dos estrategias se van a emplear, creando así dos variaciones del agente:
    - *Random*: El agente jugará hasta el final de la partida realizando movimientos al azar.
    - *Greedy*: El agente jugará los siguientes Y turnos empleando la estrategia de escoger siempre el movimiento que la función de evaluación considere como el mejor. Se ha optado por esta solución debido a lo costoso que es simular tantas partidas hasta su término y porque dichas partidas simuladas seguramente no se parecerán en absoluto a la partida real.

- *Round Robin*: en cierto modo similar al algoritmo anterior, con la diferencia de que todos los nodos serán explorados un mismo número de veces. Aplicaremos las mismas políticas con él que con el algoritmo de Montecarlo, emplearemos el filtro a los X mejores movimientos y desarrollaremos las mismas dos variaciones para este agente: *Random* y *Greedy*.

Capítulo 4
Desarrollo

En éste capítulo describiremos todo el proceso realizado desde los primeros estudios preliminares hasta obtener el sistema final.

En primer lugar hablaremos de la parte de los casos de uso identificados y de los requisitos que han sido definidos para la aplicación. Posteriormente expondremos el proceso de diseño, centrándonos especialmente en los pasos e iteraciones que nos han llevado a la función de evaluación que hemos terminado utilizando. Por último, discutiremos la implementación.

4.1. Análisis

Nuestro sistema es sencillo. Es un juego donde la interacción entre el usuario y la partida está muy definida y limitada. El juego genera bolas al azar, las coloca sobre el tablero. El jugador escoge una bola y decide la posición de destino. El juego ejecuta el movimiento tras comprobar que éste es válido, comprueba si se ha realizado línea, eliminándola si la hay o introduciendo las nuevas bolas al azar si no. La partida continuará así hasta su fin, cuando el tablero acabe finalmente lleno. Aunque por simplicidad se haya decidido integrar todo dentro de una misma aplicación, hemos dividido los casos de uso en jugador y juego.

4.1.1 Casos de uso del jugador

El jugador tiene principalmente dos interacciones con el juego: iniciar la partida y jugarla. Como hemos decidido hacer que las partidas sean configurables, el jugador también debe ser capaz de establecer los parámetros de la partida.

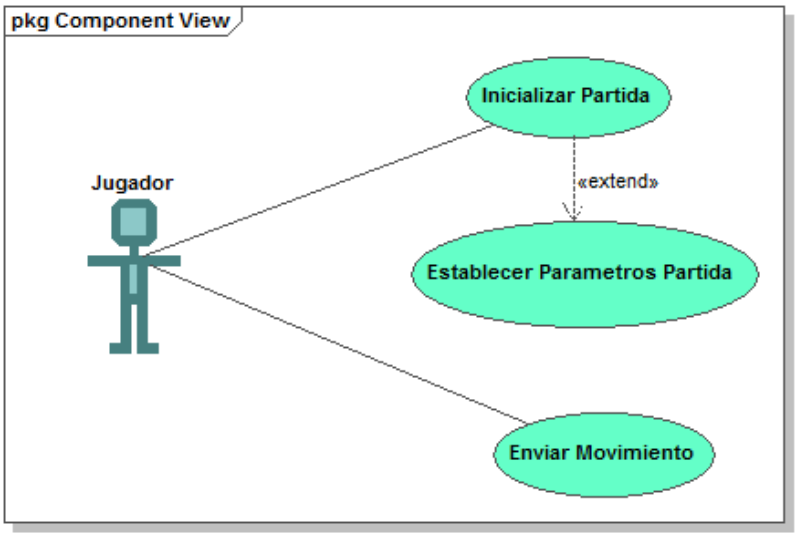

Ilustración 5 - Casos de uso Jugador

Caso de uso	**Inicializar Partida**
Resumen	El Jugador solicita Iniciar la partida
Actores	Jugador
Precondiciones	--
Descripción	El jugador inicia la partida. Escoge el tipo de jugador deseado entre humano y los agentes disponibles y decide si ejecutar la partida estándar o no. En el primer caso, la partida tendrá los siguientes parámetros: - Tablero de 9x9 - 7 colores posibles - 3 bolas cada turno que aparecerán los turnos en los que no se haga línea - Líneas de longitud mínima 5 Si elige continuar, se ejecuta el caso de uso *Establecer Parámetros Partida*
Excepciones	--
Postcondiciones	El juego comienza con los parámetros estándar o se ejecuta *Establecer Parámetros Partida*

Tabla 2- Caso de uso *Iniciar Partida*

Caso de uso	**Establecer Parámetros Partida**
Resumen	El jugador establece con qué parámetros quiere jugar la partida
Actores	Jugador
Precondiciones	El jugador decide no jugar una partida estándar.
Descripción	El jugador selecciona los parámetros que desea para la partida, escogiendo: - La dimensión del tablero (*MxN*) - El número de colores - El número de bolas que aparecerán los turnos en los que no se haga línea - Longitud mínima para hacer línea
Excepciones	--
Postcondiciones	El juego comienza con los parámetros establecidos

Tabla 3 – Caso de uso *Establecer Parámetros Partida*

Caso de uso	**Enviar Movimiento**
Resumen	El jugador envía al juego el movimiento que quiere realizar

Actores	Jugador
Precondiciones	El juego ha terminado de hacer todas las acciones necesarias tras el último movimiento válido.
Descripción	El jugador elige una casilla de origen y una casilla de destino para la cual desea ejecutar el movimiento.
Excepciones	El movimiento seleccionado no es válido.
Postcondiciones	El juego ejecuta el movimiento y realiza los cálculos pertinentes.

Tabla 4 – Caso de uso *Enviar Movimiento*

4.1.1 Casos de uso del juego

El juego por su parte tiene las siguientes responsabilidades: tiene que crear la partida con las reglas que el jugador ha establecido y mantener a éste último informado del estado actual de la partida después de cada turno. También es el encargado de comprobar la validez del movimiento que el jugador quiere realizar antes de ejecutarlo sobre el tablero y comprobar si se ha realizado línea después de realizar el movimiento, limpiando las bolas apropiadas en caso afirmativo. En caso contrario deberá introducir en el tablero las bolas del próximo turno y generar las siguientes. Por último, una vez finalizada la partida, informará al jugador de las líneas realizadas.

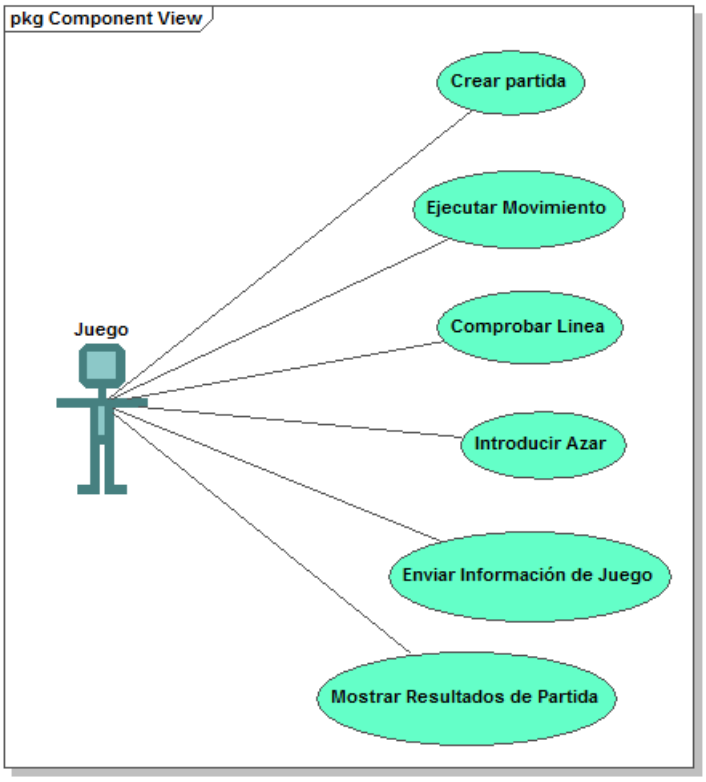

Ilustración 6 - Casos de uso Juego

Caso de uso	**Crear Partida**
Resumen	El juego crea la partida
Actores	Juego
Precondiciones	El jugador inicia la partida
Descripción	El juego crea la partida con los parámetros que el jugador ha escogido. Inicializa la partida colocando las primeras bolas sobre el tablero y ejecuta

	Enviar Información de Juego.
Excepciones	--
Postcondiciones	Enviar la información del estado inicial de la partida al jugador y se queda en espera de que el jugador seleccione su primer movimiento

Tabla 5 – Caso de uso *Crear Partida*

Caso de uso	**Enviar Información de Juego**
Resumen	El juego envía la información del estado de la partida al jugador.
Actores	Juego
Precondiciones	La partida se ha inicializado o no ha finalizado tras el último movimiento del jugador
Descripción	Tras inicializar la partida, o tras el último movimiento que no ha puesto fin a la misma, el juego recopila la información del estado de la misma y se la envía al jugador para que éste pueda seleccionar el movimiento que quiere a continuación.
Excepciones	--
Postcondiciones	El juego se queda a la espera de que el jugador seleccione otro movimiento.

Tabla 6 – Caso de uso *Enviar Información de Juego*

Caso de uso	**Ejecutar Movimiento**
Resumen	El juego ejecuta el movimiento seleccionado por el jugador
Actores	Juego
Precondiciones	El jugador envía un movimiento
Descripción	El juego comprueba que el movimiento es válido y lo ejecuta, pasando a *Comprobar Línea*
Excepciones	El movimiento no es válido por alguna de las siguientes razones: - La posición de origen no está ocupada por una bola y/o la posición de destino no está vacía - No hay un camino válido entre el origen y el destino del movimiento seleccionado El juego no ejecuta el movimiento, pidiendo al jugador que seleccione otro
Postcondiciones	El juego ejecuta el movimiento y comprueba cómo proceder antes de finalizar el turno y solicitarle otro movimiento al jugador.

Tabla 7 – Caso de uso *Ejecutar Movimiento*

Caso de uso	**Comprobar Línea**
Resumen	El juego comprueba si ha habido línea
Actores	Juego
Precondiciones	El jugador envía un movimiento válido
Descripción	El juego comprueba si se ha realizado línea tras el movimiento solicitado por el jugador. Si no la hay, el juego ejecutará *Introducir Azar*.
Excepciones	Hay línea. El juego elimina las bolas que conforman esa línea, actualiza los resultados de la partida, y ejecuta *Enviar Información*
Postcondiciones	El juego ejecuta *Introducir Azar*.

Tabla 8 – Caso de uso *Comprobar Línea*

Caso de uso	**Introducir Azar**
Resumen	El juego introduce azar en la partida
Actores	Juego
Precondiciones	El jugador envía un movimiento que no ha realizado línea
Descripción	El juego introduce al azar en el tablero las bolas que estaban previstas, y

	genera al azar bolas nuevas para la próxima vez que se necesiten, actualizando las estadísticas de la partida.
Excepciones	El número de casillas vacías libres es igual o menor que el número de bolas que hay que introducir en el tablero. El juego da por finalizada la partida y ejecuta *Mostrar Resultados de Partida*.
Postcondiciones	El juego ejecuta *Enviar Información de Juego*.

Tabla 9 – Caso de uso *Introducir Azar*

Caso de uso	**Enviar Información de Juego**
Resumen	El juego muestra las estadísticas de la partida al jugador
Actores	Juego
Precondiciones	El juego ha dado la partida por finalizada
Descripción	El juego termina la partida y le muestra al jugador las estadísticas de la misma. Estas estadísticas son: - Cuantas bolas de cada color han aparecido en el juego. - Cuantas líneas se han ejecutado durante la partida. - El número de turnos que la partida ha durado - La cantidad de bolas que se han eliminado.
Excepciones	--
Postcondiciones	El juego llega a su fin

Tabla 10 – Caso de uso *Enviar Información de Juego*

4.2. Requisitos

Los casos de uso del punto anterior nos sirven para identificar parte de los siguientes casos de usos. Otros no vienen recogidos en ellos.

Los requisitos los podemos dividir en dos fundamentalmente: Requisitos del Juego y Requisitos de los agentes.

4.2.1 Requisitos del Juego

RJ-001 – Reglas del Color Lines
Salvo en el sistema de puntuación, el juego debe comportarse exactamente como el *Color Lines*.

Tabla 11 – Requisito *RJ-001*

RJ-002 – Elementos configurables
Al comenzar la partida, se le pedirá al jugador que establezca los elementos que van a definir la partida. Estos elementos son: • Número de filas • Número de columnas • Número de colores • Longitud de la línea • Bolas que aparecen por turno

Tabla 12 – Requisito *RJ-002*

RJ-003 – Resultado de la partida
A lo largo de la partida, el juego irá guardando información sobre los siguientes elementos: • Número de líneas realizadas • Número total de bolas eliminadas

• Número de bolas de cada color que han aparecido

Tabla 13 – Requisito *RJ-003*

RJ-004 – El juego no poseerá información que no posea el jugador
La información que ve el jugador será exactamente la misma que el juego posea. Esto implica, fundamentalmente, que el juego no decidirá donde colocar las bolas nuevas, ni cuáles serán el reemplazo de éstas, hasta que se resuelva el movimiento que el jugador envía.

Tabla 14 – Requisito *RJ-004*

4.2.4 Requisitos de los Agentes

RA-001 – Independencia del juego
El agente y el juego sólo se mandarán entre sí la información necesaria para el transcurso de la partida. El juego mandará la información del estado de la partida (tablero y próximas bolas), y el jugador enviará sólo el movimiento que haya decidido realizar.

Tabla 15 – Requisito *RA-001*

RA-002 – Independencia de la partida
Los agentes deben estar preparados a nivel de código para jugar cualquier tipo de partida posible, sin importar la configuración de ésta.

Tabla 16 – Requisito *RA-002*

RA-003 – Evaluación del Estado
La función de evaluación deberá ser rápida en evaluar un estado del tablero, tomándose no más de un milisegundo para ello.

Tabla 17 – Requisito *RA-003*

RA-004 – Velocidad del Greedy
Teniendo en cuenta de que el agente *Greedy* es el que podemos utilizar para ver el funcionamiento de la función de evaluación, éste agente debería tomarse no más de cinco minutos para jugar una partida completa (teniendo dicha partida la configuración por defecto).

Tabla 18 – Requisito *RA-004*

RA-005 – Puntuación del Greedy
Por los mismos motivos que en el requisito *RA*-004, el agente *Greedy* deberá realizar una media mínima de 40 líneas por partida antes de considerarse la función de evaluación como válida. Éste número se ha escogido al ser una puntuación razonable para un jugador humano novato.

Tabla 19 – Requisito *RA-005*

4.3. Diseño

En esta sección mostraremos los diagramas de clases de la aplicación, comentando un poco el contenido o la función que cada clase cumple.

4.3.1 Five Lines

En esencia la aplicación, excluyendo clases auxiliares que encapsulan o bien datos o bien alguna funcionalidad utilizada por varias clases, se puede resumir con el siguiente diagrama.

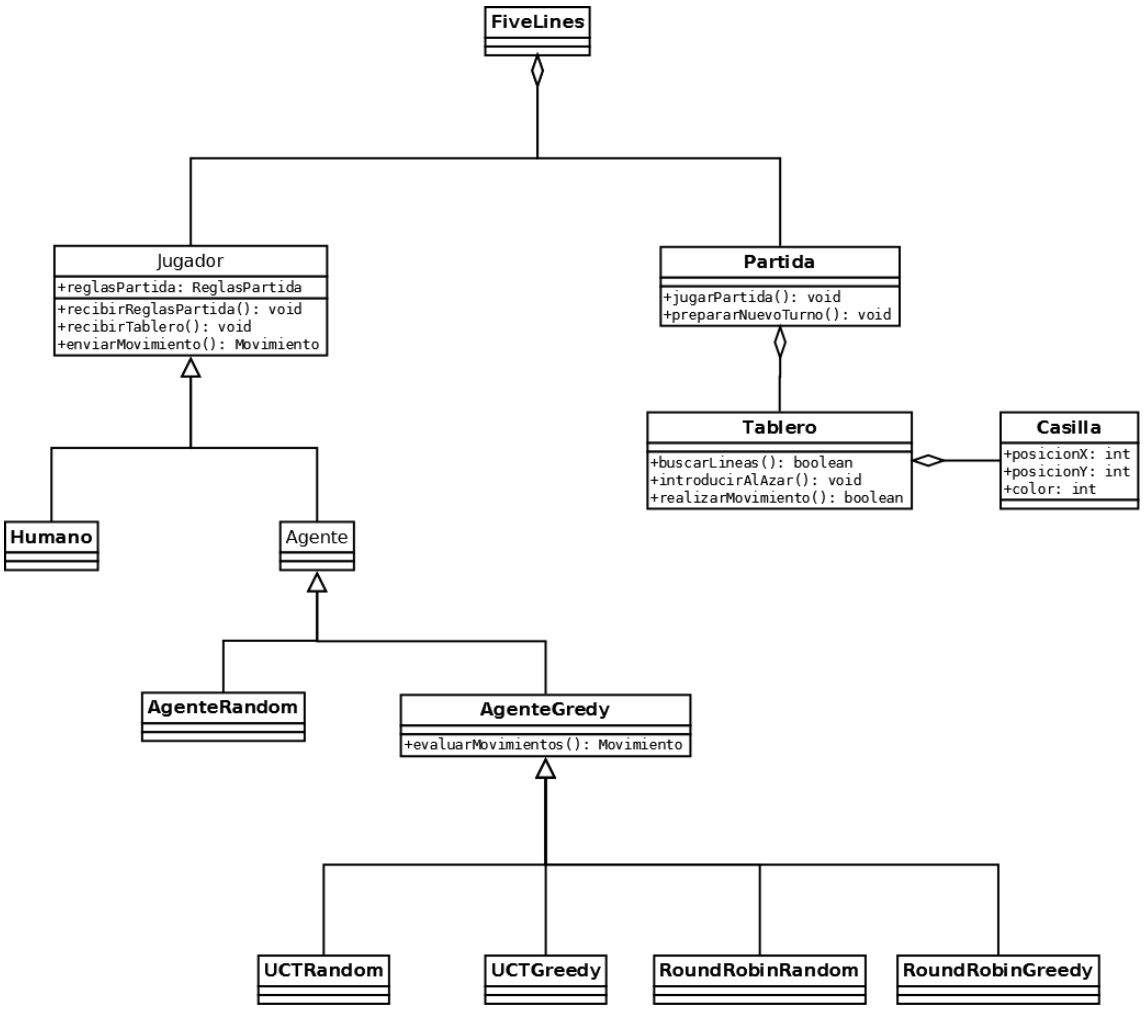

Ilustración 7 – Diagrama de clases de *Five Lines*

Cada una de estas clases cumple una función determinada:

- **FiveLines:** Clase principal e inicio de la ejecución del programa.
- **Partida:** Clase responsable del motor del juego y responsable de mantener el Tablero real de la partida, asegurándonos que no se modifica de forma externa. Se encarga de enviar la información de la partida al jugador y comunicarle al tablero los movimientos que éste desea ejecutar. Solicitará al tablero que compruebe si se ha realizado línea o no y generará los colores de las bolas que deben aparecer el siguiente turno. Por último, es la clase responsable de contabilizar los resultados y estadísticas de la sesión de juego actual.
- **Tablero:** Clase que representa el tablero de juego actual, exceptuando las bolas que aparecerán próximamente. Es el encargado de buscar líneas cuando se le solicite, y de introducir las bolas en posiciones vacías al azar. También comprueba si el movimiento a ejecutar es un movimiento válido o no.
- **Casilla:** Clase que encapsula una casilla en el tablero. Guarda información sobre sus coordenadas, sobre el color de la bola que está ocupándola (o cero si está vacía),

además de información auxiliar para facilitar algunas de las funciones del juego, como puede ser el identificador de zona para casillas vacías (que ayuda a identificar movimientos válidos como explicaremos más adelante).

- **Jugador:** Clase abstracta que implementa algunos métodos comunes para todos los jugadores, sean éstos humanos o agentes. Encargada de la comunicación con la Partida, recibiendo las reglas y el estado de ésta (una copia exacta del tablero y las bolas que aparecerán en el turno próximo). Obliga a sus clases hijas a implementar el método *enviarMovimiento()*.
- **Humano:** Implementa la lógica para cuando el jugador es un humano. Su método enviar movimiento informa por pantalla del estado actual de la partida y le da las instrucciones necesarias para enviar el movimiento al juego.
- **Agente:** Clase abstracta que implementa algunos métodos comunes para todos los agentes, como calcular todos los movimientos posibles para un tablero dado y algunas estructuras de datos comunes.
- **AgenteRandom:** Representa al jugador aleatorio. Simplemente escoge uno de los movimientos posibles y lo envía.
- **AgenteGreedy:** Clase que implementa la función de evaluación. Su estrategia es analizar todos los movimientos posibles y enviar el mejor de todos ellos. El motivo por el cual los siguientes agentes heredan de ella es porque todos utilizan sus decisiones como filtros para evitar iterar sobre los movimientos peores. Para realizar su cometido, ésta clase se sirve de las clases *TableroConectividad* y *TableroZonas*, las cuales se explicarán en la siguiente sección.
- **AgenteUCTRandom:** Este agente implementa el algoritmo UCT seleccionando primero los *X* mejores movimientos según el *Greedy* y luego iterando sobre ellos siguiendo el algoritmo UCT. Los siguientes nodos, para cada uno de los nodos conseguidos tras el filtro, los explorará escogiendo al azar un movimiento de entre los posibles, continuando esta ejecución hasta llegar al final de la partida. Una clase auxiliar le puntuará el tablero final. Para ejecutar la partida asegurándonos de que no se interfiere con el tablero real, utiliza la clase *SimuladorPartida* para llevar la partida a término. También se sirve de la clase *TableroFinal* para puntuar el resultado de la simulación. Ambas clases se explicarán en la siguiente sección.
- **AgenteUCTGreedy:** Este agente, al igual que el anterior, filtra los mejores movimientos de acuerdo con el *Greedy* para iterar sobre ellos. Tras obtener los nodos iniciales a puntuar, el resto de movimientos los escogerá siguiendo la misma política que el agente *Greedy*, jugando el mejor de los posibles. Este procedimiento se repetirá hasta llegar a una profundidad igual a tres, ya que se consideró que más lejos el Tablero dejaría de resultar representativo. Una vez llegados a este punto, una clase auxiliar le puntuará el tablero resultante. Al igual que la clase *AgenteUCTRandom* emplea las clases *SimuladorPartida* y *TableroFinal*.
- **AgenteRoundRobinRandom:** Similar al AgenteUCTRandom, pero explorando los movimientos utilizando la política *Round Robin*.
- **AgenteRoundRobinGreedy:** Similar al AgenteUCTGreedy, pero explorando los movimientos utilizando la política *Round Robin*.

4.3.2 Clases Auxiliares

Además de éstas clases, que representan la estructura fundamental de la aplicación, se emplean las siguientes clases auxiliares.

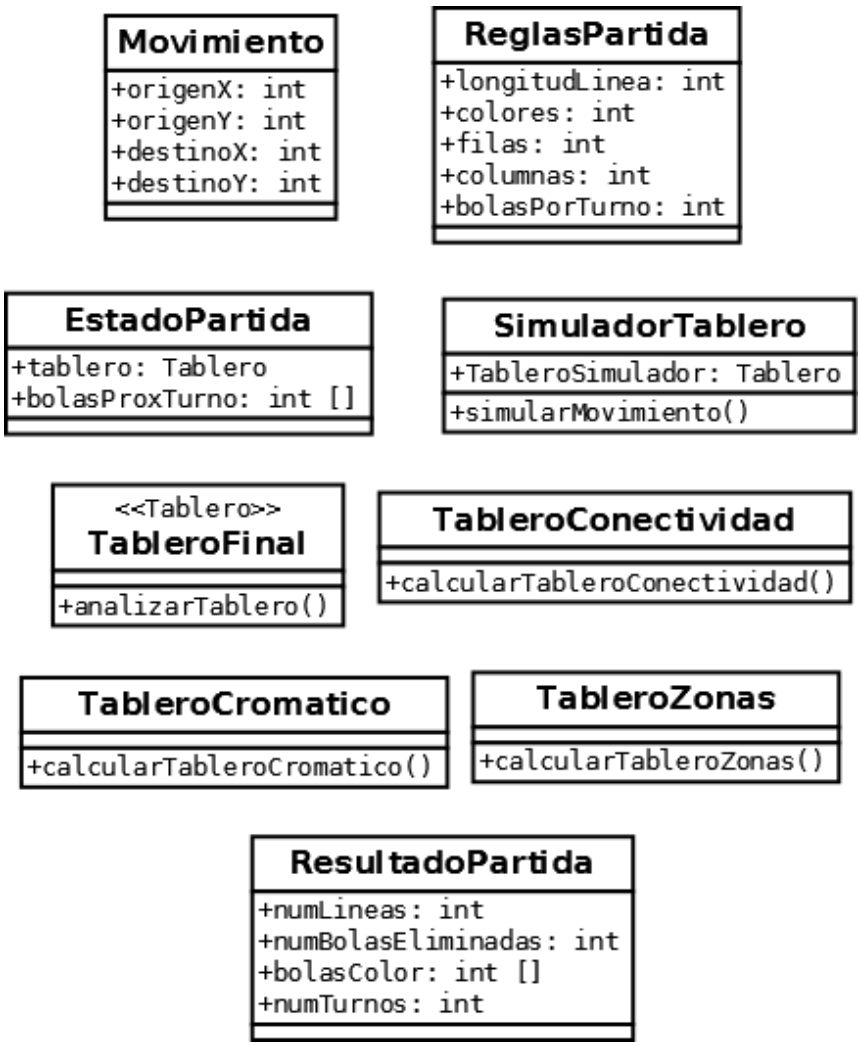

Ilustración 8 - Clases Auxiliares de *Five Lines*

- **Movimiento:** Esta clase encapsula los movimientos que se pueden hacer. Es simplemente una estructura de datos que guarda la posición de origen y de destino del movimiento en cuestión.
- **ReglasPartida:** Esta clase es simplemente un almacén de datos sobre la configuración de la partida que se está jugando actualmente. Otras clases la consultan cuando necesitan consultar una característica en concreto, como el número de columnas, o cuántos colores hay.
- **EstadoPartida:** Esta clase es un *snapshot* del estado actual de la partida, es decir, cuál es el estado del tablero y cuáles son las siguientes bolas que aparecerán en él próximamente.
- **SimuladorPartida:** Esta clase ayuda a los agentes que implementan *Round Robin* y *UCT* a jugar turnos siguientes asegurándonos que la integridad de los datos

gestionados por la clase *Partida* se mantienen intactos. En esencia funciona de un modo similar a la clase *Partida*.

- **TableroFinal:** Esta clase ayuda a los agentes que implementan *Round Robin* y *UCT* a analizar el tablero resultante una vez han terminado de simular los movimientos, otorgando una puntuación en función de la distribución de las bolas y, si aplica, la cantidad de casillas vacías restantes.
- **TableroCromatico:** Esta clase analiza la distribución de los colores para un tablero dado. En esencia son varios tableros, uno por color (tratando el resto de colores como obstáculos). Se ofrecerá más información sobre éste tablero en la sección de Implementación.
- **TableroConectividad:** Esta clase analiza la conectividad de cada una de las casillas para un tablero determinado. Ésta funcionalidad es la más importante de la función de evaluación y, por lo tanto, se explicará en la sección de Implementación.
- **TableroZonas:** Esta clase analiza las "zonas" que hay un tablero determinado. Por "zona" entendemos una colección de casillas vacías que están conectadas entre sí horizontal o verticalmente.
- **ResultadoPartida:** Almacén de las estadísticas de la partida actual. Almacena el número de turnos jugados, el número de líneas realizadas, el número de bolas eliminadas y cuántas bolas de cada color han ido apareciendo hasta el momento.

4.4. Implementación

Esta sección está dedicada a explicar, sin entrar en demasiado detalle con los particulares, cómo se ha implementado cada uno de los componentes de la aplicación. Comenzaremos mencionando las tecnologías empleadas en el desarrollo y después hablaremos de los componentes.

4.4.1 Tecnologías

A continuación hablaremos brevemente de las tecnologías empleadas para el desarrollo de nuestra aplicación

4.4.1.1 Java

Java es un lenguaje de programación orientada a objetos, diseñado por *Sun Microsystems* con la idea de tener tan pocas dependencias de implementación como fuesen posibles. Aunque deriva de *C* y *C++* en cuanto a sintaxis, Java ofrece menos utilidades de bajo nivel que ellos. El código desarrollado en Java es, en teoría, independiente de la máquina en la que se ejecuta, ofreciendo gran portabilidad entre sistemas. Sin embargo, debido a que sus ejecutables no son exactamente código máquina, sino que la JVM (*Java Virtual Machine*) debe interpretarlos según el sistema en el que se encuentre, Java es un lenguaje bastante pesado de ejecutar. Esto siempre ha sido una de las mayores críticas a las que el lenguaje se ha enfrentado desde su concepción.

La razón por la que se ha escogido Java es porque, casi desde su concepción, fue, y sigue siendo uno de los lenguajes de programación más populares, especialmente en lo que programación orientada a objetos se refiere.

4.4.1.2 NetBeans

Netbeans es un entorno de desarrollo integrado de código abierto fundado por *Sun Microsystems* y hecho especialmente para programar en Java. Aunque existen varias formas de ampliarlo, para éste proyecto hemos utilizado la versión base 7.0, sin ningún *addon* o *plugin*.

Netbeans, junto con Eclipse, es uno de los entornos de desarrollo para Java más populares y, pese a estar codificado completamente en Java, es bastante ligero, lo que hace que sea una plataforma ideal para desarrollos pequeños.

4.4.1.2 Subversion

Subversion (SVN) es un sistema de control de versiones de código abierto mantenido por Tigris. Para este proyecto se ha instalado en una segunda máquina, conectada en la misma red local que la máquina de desarrollo, un pequeño repositorio de Subversion para ejercer tanto de sistema de control de versiones como de *backup*. Además, Subversion ha sido útil para identificar cambios que han causado problemas y volver a la última versión estable en esos casos.

4.4.2 Juego

La implementación del juego en sí recae, fundamentalmente, en las siguientes cuatro clases:

- **Partida:** Esta clase es la encargada de gestionar el conjunto de reglas del juego, es decir, mantener la estructura de cada uno de los turnos, gestionar la comunicación con el jugador y detectar si la partida se ha terminado.
- **Tablero:** Clase que contiene la información del tablero actual, y responsable de detectar líneas y decidir si un movimiento es o no válido.
- **Casilla:** Clase que encapsula toda la información de la casilla, incluyendo su posición dentro del tablero y su propio estado, es decir, si hay o no bola y de qué color es ésta en caso afirmativo.
- **Movimiento:** Clase que encapsula la posición de origen y la posición de destino para un movimiento potencial (sea o no válido).

4.4.2.1 Conjunto de reglas

El conjunto de reglas del Five Lines es bastante sencillo de implementar asegurándonos que todas sus comprobaciones se realizan en el momento oportuno, principalmente por tratarse de un juego por turnos. Tras crear la partida e introducir las primeras bolas en posiciones aleatorias, la clase *Partida* debe hacer lo siguiente:

1. Envía la información de la partida al jugador.
2. Solicita al jugador que envíe un movimiento y se queda a la espera hasta que lo recibe.
3. Una vez recibido el movimiento del jugador, le pide al Tablero que compruebe si es un movimiento legal.
4. Si no lo es, vuelve al paso 2.
5. Tras haber comprobado que el movimiento es válido, le pide al Tablero que lo ejecute. Acto seguido le solicita que detecte si ha habido líneas.

   a. Si las ha habido, se asegura de que las bolas que se encuentran en una línea se han eliminado y vuelve al paso 1.
   b. Si no las ha habido, comprueba si el introducir nuevas bolas pondría fin a la partida. Si no es el caso, le envía la información de las bolas que hay que introducir al azar al Tablero para que éste se encargue de ello y después genera al azar la nueva colección de bolas a introducir el siguiente turno en el que no se forme una línea. Una vez terminados estos dos procesos se volverá otra vez al paso 1.
6. Si las nuevas bolas pondrían fin a la partida se le comunica al jugador informándole además de la puntuación obtenida.

4.4.2.2 Comprobación de la validez de un movimiento

Los movimientos de bolas de una casilla a otra proporcionan las únicas restricciones que el juego le impone al jugador. Estas restricciones son las siguientes:

1. La casilla de destino debe estar vacía. Es decir, no se pueden intercambiar las posiciones de dos bolas.
2. El camino entre la casilla ocupada de origen y la casilla vacía de destino deberá estar compuesto de una secuencia de casillas vacías conectadas entre sí horizontal y/o verticalmente. Casillas conectadas en diagonal no forman un camino.

Inicialmente, tras la comprobación de la primera restricción, verificábamos que el camino fuese válido buscando por recursividad dicho camino hasta agotar todas las posibilidades o lo encontrábamos. Sin embargo, durante las primeras pruebas con los agentes detectamos que esta opción no era lo suficientemente buena en términos de rendimiento. Para solucionarlo, diseñamos el concepto de “zonas”.

Entendemos en éste juego por zona como una colección de casillas vacías unidas entre sí bien horizontalmente, bien verticalmente. Para distinguir unas de otras, asignamos a cada casilla vacía un valor numérico que refleja la zona en la que se halla. Éstas zonas se recalculan tras resolver la ejecución de movimientos nuevos (limpiar una línea o introducir bolas nuevas).

Para calcular una zona, utilizamos el siguiente algoritmo:

1. Recorremos el tablero hasta encontrar la primera casilla vacía a la que no se le ha asignado zona.
2. Una vez encontrada, le asignamos la zona actual y buscamos en las cuatro posiciones válidas (arriba, abajo, izquierda y derecha) la existencia de otra casilla vacía anexa.
3. Si encontramos una casilla vacía, repetimos el paso 2 con la nueva casilla.
4. Finalizado el paso 3, incrementamos el número de la zona actual y repetimos el proceso desde el paso 1. Esto seguirá así hasta recorrer todas las casillas.

Pese a parecer a priori un proceso costoso, éste nos permite agilizar en gran medida la comprobación de la validez de un movimiento concreto, ya que lo único que debemos mirar es si una bola tiene conexión con la zona donde se encuentra la casilla determinada. Es decir, si la bola tiene inmediatamente a su lado (en horizontal o en vertical) una casilla vacía cuyo número

de zona coincide con la zona en la que se encuentra la casilla vacía de destino, entonces el movimiento es válido.

Supongamos el siguiente ejemplo:

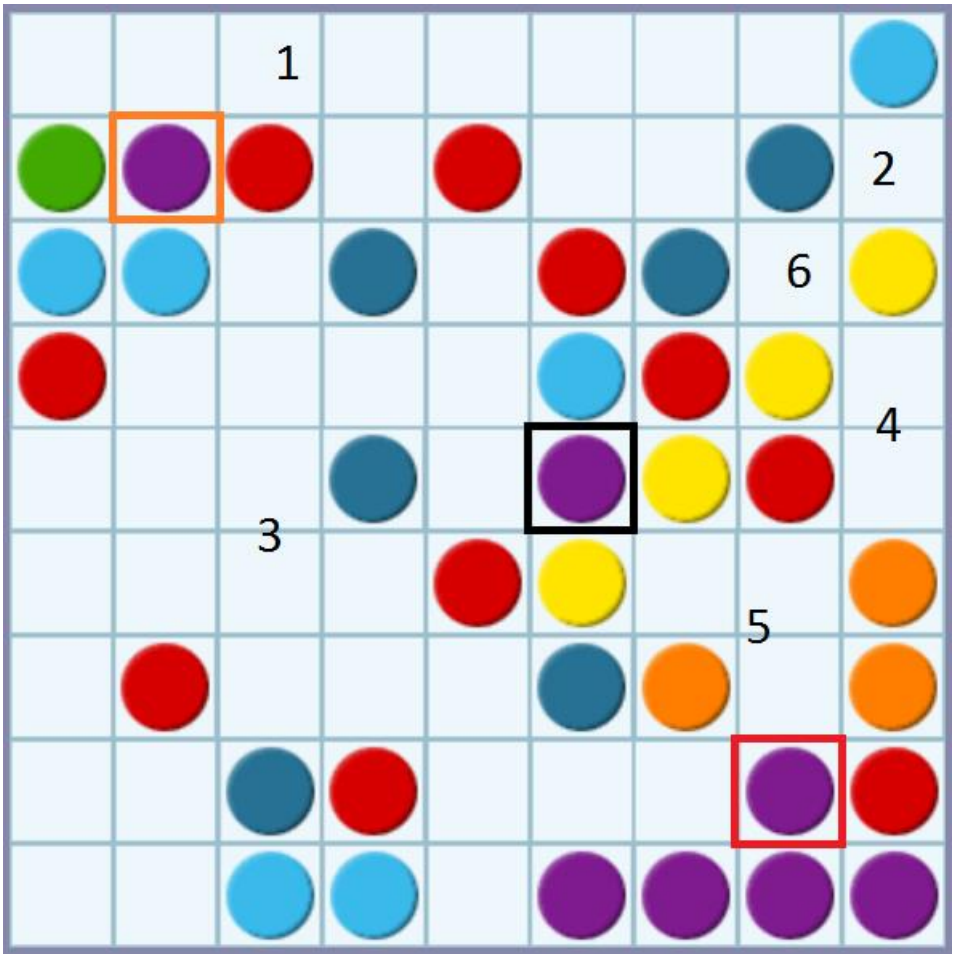

Ilustración 9 – Zonas de movilidad en el tablero

Para poder completar la línea morada de la parte inferior hay tres bolas moradas candidatas no usadas para la línea, marcadas en los colores naranja, negro y rojo. La que está marcada en naranja sólo tiene acceso a la zona 1, mientras que la casilla a la que tenemos como destino se encuentra en la zona 3. Tanto la marcada en negro como la marcada en rojo nos valdrían para completar la línea, ya que ambas tienen acceso a la zona 3 y, por lo tanto, pueden desplazarse a cualquier casilla vacía que se encuentre en ella.

Si bien para un jugador humano ambos métodos son igualmente eficientes, en el sentido de que no apreciará demora durante el cálculo del primer método, con los agentes nos ahorramos mucho tiempo de cómputo cuando tenemos que calcular todos los movimientos posibles antes de empezar a evaluarlos.

4.4.2.3 Búsqueda de líneas

A continuación trataremos brevemente el mecanismo para buscar líneas en el tablero una vez se ha ejecutado el movimiento, principalmente porque éste es el objetivo fundamental del jugador.

Éste mecanismo no es particularmente complicado. Una vez se ha ejecutado el movimiento, se le solicita al tablero que busque líneas. Las líneas pueden estar en horizontal, en vertical o en cualquiera de las dos diagonales. El proceso, para cada una de estas direcciones, es el siguiente:

1. Seleccionamos la primera casilla ocupada que no se haya visitado ya.
2. Comprobamos si, para la dirección en la que estamos mirando, la siguiente bola es del mismo color que la casilla anterior.
3. Si lo es, repetimos el paso 2 usando como base la nueva casilla y decrementando la longitud objetivo.
4. Una vez encontrado una bola de un color diferente, miramos si hemos llegado a cero o menos en la longitud objetivo. Si lo hemos conseguido, marcamos todas las casillas visitadas en los pasos 2 y 3 como que están en línea y volvemos al paso 1.
5. Repetimos este proceso hasta llegar al final del tablero.

Una vez repetido este algoritmo para las cuatro direcciones posibles, sabremos mediante un *flag* si hemos obtenido línea o no. En caso afirmativo, simplemente tendremos que vaciar las casillas que estén marcadas para ello.

4.4.2.4 Generación del azar

Por último, hablaremos brevemente del mecanismo de introducción del azar en la partida, puesto que éste es el mayor obstáculo entre el jugador y su objetivo.

Si tras ejecutar el movimiento no se ha encontrado línea, la partida le pide al tablero, el cual mantiene una lista de qué casillas están vacías, que introduzca las nuevas bolas. El tablero seleccionará de la lista tres casillas al azar y las marcará como ocupadas con el color también elegido al azar. Una vez hecho esto, la partida generará aleatoriamente las nuevas bolas a introducir.

4.4.3 Función de evaluación

En éste apartado hablaremos de la función de evaluación empleada para enfrentarnos a este juego y de las diferentes iteraciones y versiones por las que hemos pasado. El objetivo fundamental de esta función de evaluación era facilitar que los agentes jugasen en modo "supervivencia", es decir, intentando conseguir línea lo antes posible, sin preocuparse demasiado por eliminar más o menos bolas con un solo movimiento.

4.4.3.1 Tableros Cromáticos

La primera aproximación a obtener una función de evaluación. Tras un primer estudio sobre el juego, identificamos que cada casilla podía, dependiendo de su posición, participar en un número *X* de líneas. Por ejemplo, en el tablero estándar, una esquina puede participar en sólo tres líneas diferentes (una horizontal, una vertical y una diagonal). Según nos vamos alejando de los bordes hacia el centro, éste número crece hasta llegar a un máximo de veinte líneas en la casilla central.

3	4	5	6	8	6	5	4	3
4	6	7	9	11	9	7	6	4
5	7	10	12	14	12	10	7	5
6	9	12	15	17	15	12	9	6
8	11	14	17	20	17	14	11	8
6	9	12	15	17	15	12	9	6
5	7	10	12	14	12	10	7	5
4	6	7	9	11	9	7	6	4
3	4	5	6	8	6	5	4	3

Ilustración 10 – Tablero de líneas distintas por casilla

La primera idea que tuvimos para desarrollar una función de evaluación fue lo que llamamos tableros cromáticos. Decidimos aplicar el concepto anterior generando un tablero por cada uno de los colores involucrados, tratando en cada uno de ellos las bolas de otros colores como obstáculos. El objetivo de ésta aproximación era intentar maximizar los valores de cada uno de los tableros, moviendo bolas de un color a posiciones ventajosas para dicho color, minimizando en lo posible los efectos negativos que pudiesen tener sobre el resto de colores. Esta aproximación premiaba, por lo tanto, Movimientos que separasen los colores en secciones diferentes del tablero.

Pese a que utilizando esta aproximación con el agente *Greedy* se conseguían líneas con bastante frecuencia, los resultados no fueron los deseados. Las partidas se cerraban obteniendo muy pocas líneas o en bastantes ocasiones ninguna. El comportamiento real de esta estrategia tendía a agrupar las bolas de cada color en los bordes y sin realmente intentar formar líneas, ya que ambos factores eran los más "beneficiosos" para el resto de colores. Las bolas quedaban agrupadas en grupos sin tener una apreciable tendencia a cerrar líneas. Se consiguió mejorar algo el comportamiento tras introducir algunos cambios sobre éste concepto, pero no mejoró lo suficiente como para que lo considerásemos una estrategia viable y, por lo tanto, se descartó.

4.4.3.1 Conectividad

Definimos la conectividad de una casilla como el número mínimo de turnos que se necesitan para realizar una línea con ella (o, mejor dicho, vaciarla) sin cambiar el contenido de dicha casilla. Cuanto menor sea este número, mejor su conectividad. Si la casilla está vacía, su conectividad será por lo tanto cero, puesto que no contiene ninguna bola.

Por ejemplo, dado el siguiente tablero:

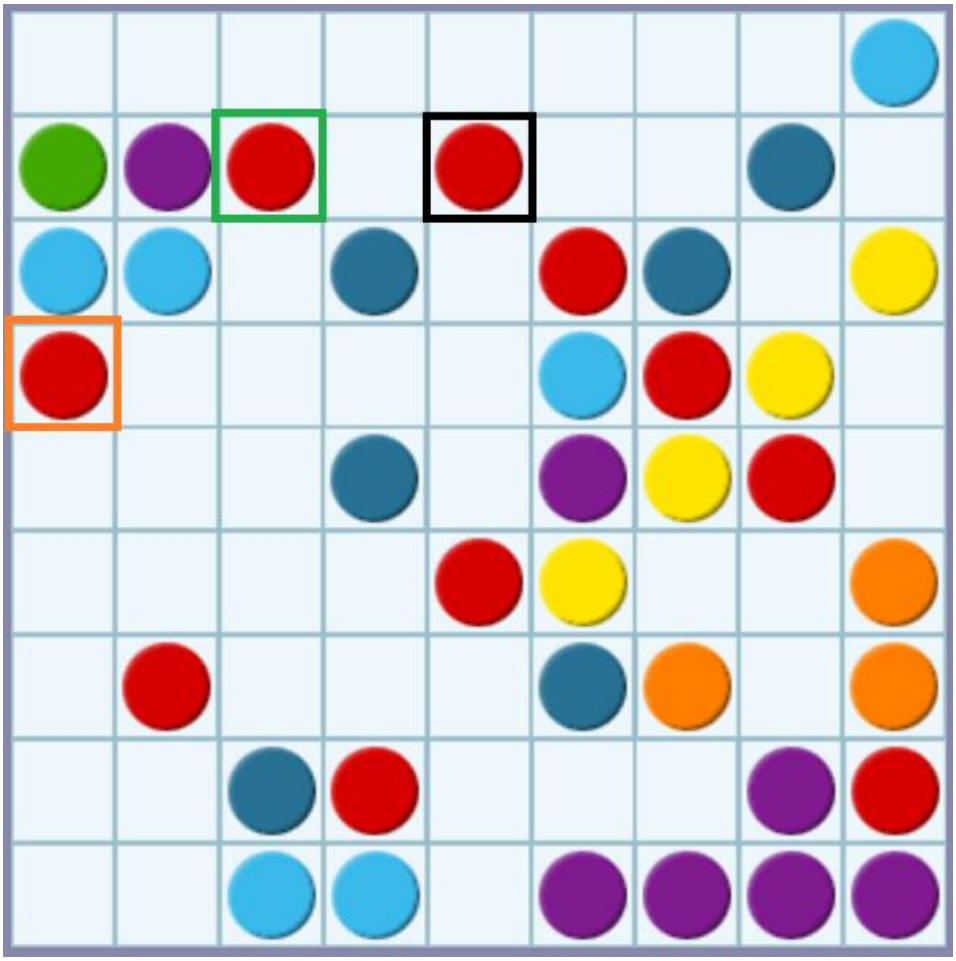

Ilustración 11 – Ejemplo de conectividad

Las conectividades de las casillas marcadas serían las siguientes:

- La casilla marcada en naranja no está en línea con ninguna bola de su color, por lo tanto se necesitarían cuatro turnos para hacer línea con ella. Conectividad 4.
- La casilla marcada en verde está en línea con otra bola de su mismo color, y hay tres espacios vacíos que permitirían completar la línea entre ellas. Por lo tanto, su conectivdad será 3.
- La casilla marcada en negro está en línea con la casilla del caso anterior, por lo que su conectividad es al menos 3. Sin embargo, está formando una línea de longitud cuatro con otras, y hay una casilla vacía que permite cerrar la línea en el próximo movimiento. Por lo tanto, tanto su conectividad como la conectividad de las bolas con las que está alineada es 1.

Esta segunda aproximación fue considerablemente más exitosa desde su primera versión, consiguiendo que el agente *Greedy* realizase bastantes líneas en todas sus partidas, del orden de treinta o cuarenta de media y, muy raramente, menos de diez. Para decidir entre un tablero u otro, éste análisis devolvía un vector contabilizando el número de casillas dentro de un valor de conectividad dado. Luego, a la hora de elegir entre dos tableros se comparaban los vectores, eligiendo el que tuviese mayor valor en las posiciones más bajas, ya que esto implica que cada vez tenemos más líneas de mayor longitud o, incluso, que el movimiento realizado finalizará una línea.

Ahora hablaremos de las diferentes versiones por la que hemos iterado, explicando las mejoras que cada una de ellas ofrece.

4.4.3.1.1 Conectividad, primera versión

En ésta primera versión, mencionada arriba brevemente, considerábamos la conectividad utilizando sólo la información que teníamos en el tablero, sin tener en cuenta las bolas que podrían caer en el siguiente turno.

La conectividad se calculaba de la siguiente forma:

1. Recorremos el tablero hasta encontrar una casilla ocupada no visitada, y establecemos una conectividad inicial de *N-1*.
2. Para una casilla ocupada y una dirección determinada, miramos la siguiente posición en la dirección que buscamos.
    a. Si tenemos una bola del mismo color, decrementamos la conectividad en 1 y repetimos éste paso.
    b. Si tenemos una bola de un color diferente o una casilla vacía, finalizamos la búsqueda, actualizamos el valor de la conectividad al resultante para las casillas visitadas.
3. Repetimos los pasos anteriores para todas las direcciones posibles y todas las casillas del tablero. Asignamos como conectividad para esas casillas la menor de las cuatro conectividades posibles.

Así pues, la conectividad es un valor que va de 0 (casilla vacía) a *N-1*, siendo *N* el número de bolas requerido para realizar una línea. Puesto que el objetivo es tratar de tener cuantas más casillas vacías, éste acercamiento promueve mucho más directamente la creación de líneas en el tablero. Si el número de casillas que tienen conectividad 0 aumenta tras ejecutar el movimiento, entonces sabemos que se ha realizado línea.

Pese a obtener buenos resultados con esta primera versión, detectamos que había varios problemas que afrontamos en versiones posteriores:

- No teníamos en cuenta en absoluto las bolas que aparecerían en el próximo turno.
- La representación de la conectividad no era del todo realista. Se asumía que cuatro bolas de un color, encerradas por “obstáculos” tenían la misma conectividad que las mismas cuatro bolas sin obstáculos.

- El mínimo número de turnos tampoco consideraba la posibilidad de que no tuviésemos accesible, ni en el turno actual ni en turnos venideros, una bola del color apropiado que nos ayudase a continuar o cerrar la línea.

Viendo estas limitaciones, y pese a tener un mecanismo de evaluación razonablemente bueno, se decidió iterar sobre ésta idea para hacerla más representativa del estado actual de la partida.

4.4.3.1.2 Conectividad, segunda versión

Para ésta versión decidimos afrontar el segundo problema, ya que lo consideramos como el mayor fallo en el concepto de conectividad. El primer problema dependía en gran medida del azar, y el tercero estaba en cierto modo ligado con el primero.

Entendemos como obstáculo para una casilla ocupada determinada, como las bolas de otro color que bloquean las posibilidades de hacer línea utilizando esa casilla. Estos obstáculos incrementarán en uno la posibilidad de hacer línea utilizando la casilla que estamos estudiando para una dirección determinada. Este estudio se prolonga hasta que encontremos el camino más corto posible hasta la primera casilla vacía.

Tengamos en cuenta, por ejemplo, la siguiente situación en el tablero:

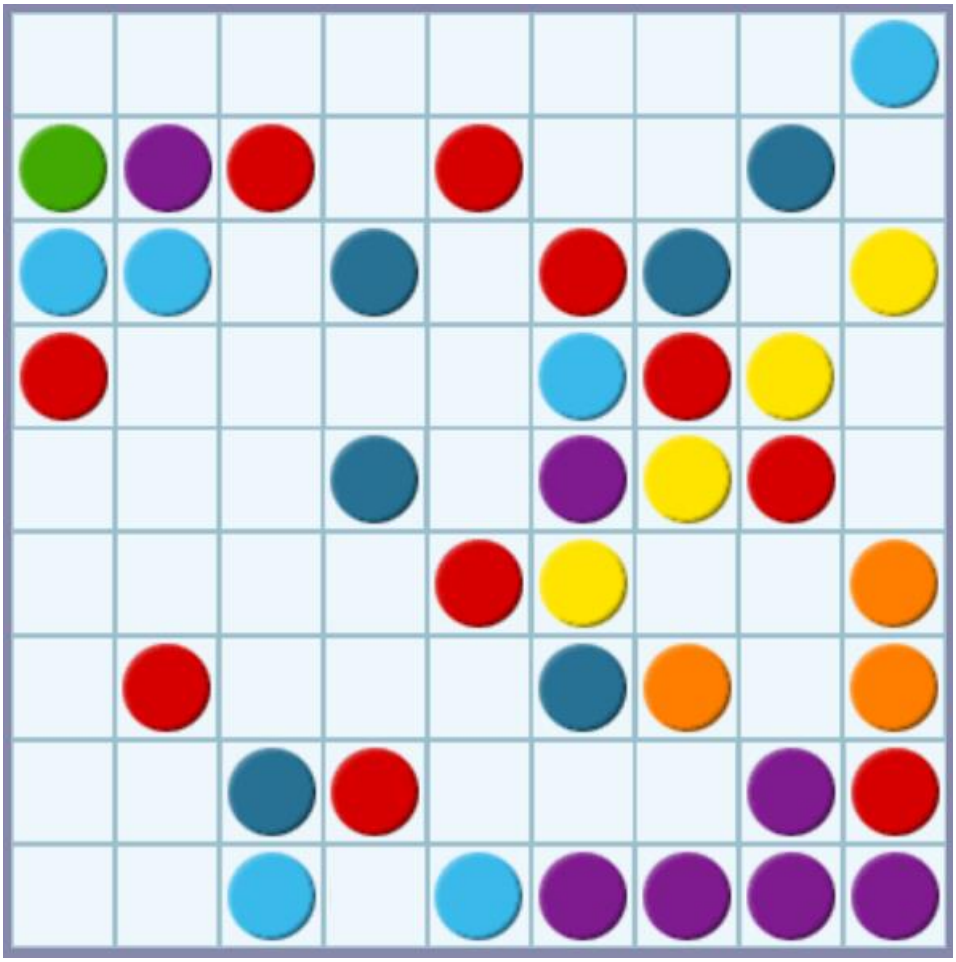

Ilustración 12 – Ejemplo de situación de tablero

En éste caso tenemos tres líneas parciales de longitud cuatro: una morada, una amarilla y una roja. Con la primera versión de la conectivdad, consideraríamos que las tres tienen una conectivdad de 1, puesto que falta sólo una bola más para cerrar la línea. Sin embargo, para el caso de la línea morada esto no es así, ya que hay un obstáculo bloqueando la única casilla que permitiría finalizarla. Por lo tanto, la conectividad de las líneas roja y amarilla es uno, mientras que la conectividad de la morada es dos, necesitando un turno adicional para mover el obstáculo.

Después de implementar estos cambios y comprobar que efectivamente mejoraban considerablemente los resultados obtenidos por el agente *Greedy* al tener éste una representación más fiel del estado del juego, pasamos a realizar la siguiente y última mejora sobre éste concepto.

4.4.3.1.3 Conectividad, tercera versión

Por último, tratando la conectividad estudiamos como utilizar las bolas que aparecerían en el turno siguiente y el hecho de si teníamos o no bolas del color apropiado a mano en el actual.

El concepto detrás de esta idea es sencillo. Suponiendo una línea formada, a falta de una sola bola para completarse y sin obstáculos que nos lo impidan; la conectividad de esa línea varía de forma un poco optimista (es decir, suponiendo que el mejor caso posible ocurrirá si se introduce algo de azar) en función de los siguientes casos:

- Tenemos ya una bola en juego que no forma parte de la línea y que está en la zona apropiada para cerrarla en el siguiente turno. La conectividad no varía, puesto que asumimos que ningún obstáculo nos impedirá moverla a la casilla destino. Conectividad 1.
- No disponemos de una bola del color apropiado que nos permita cerrar la línea, pero vemos que una bola del color que buscamos aparecerá en el próximo turno. Pese a que una aproximación cien por cien optimista nos diría que éste caso es igual que el primero (es decir, la bola caerá en la zona apropiada y, por lo tanto requeriremos el mismo número de turnos para formar línea que en el caso anterior), la lógica nos indica que éste no es el caso (la bola puede caer en otra zona) y, por lo tanto, incrementamos en uno la conectividad de la línea. Conectividad 2.
- No disponemos de una bola del color apropiado que nos permita cerrar la línea, y vemos que en el turno próximo no aparecerá ninguna bola de dicho color. Éste es el peor caso, pero somos optimistas, así que asumimos lo mejor y aumentamos la conectividad en dos. Conectividad 3.

Por ejemplo, para el caso expuesto en la sección anterior, tenemos las siguientes conectividades:

- La línea roja tiene una bola de su color con un camino válido a una casilla que le permite completar la línea con el próximo movimiento. Conectividad 1.
- La línea morada tiene un obstáculo en medio pero, una vez movido, dispondría, suponiendo que no aparezcan obstáculos que lo impidan, de dos candidatas para cerrar la línea. Conectividad 2.

- Con la línea amarilla, en las dos versiones de la conectividad anteriores tendríamos una conectividad de 1. Ahora dependerá de las bolas que aparezcan en el turno siguiente:
    - Si viene una bola amarilla, con suerte ésta caerá en algún punto del tablero que nos permita mover la bola a la casilla apropiada y completar la línea, requiriendo el mismo número de turnos que para completar la línea morada y, por lo tanto, su conectividad será 2.
    - Si no viene, entonces necesitaremos esperar, al menos, un turno más para que aparezca. Su conectividad será 3.

Este cambio menor también ayudó bastante a mejorar los resultados obtenidos. Pese a que consideramos que el concepto de la conectividad podría mejorarse aún algo más, la aproximación que teníamos era lo suficientemente buena. Sin embargo, para ir un poco más lejos, decidimos estudiar posibles mecanismos de desempate para las situaciones en las que la función de evaluación nos dictaba que dos o más movimientos eran igualmente buenos, en lugar de ejecutar siempre el primero que se encontrase o elegir uno de ellos al azar.

4.4.3.3 Conectividad y tableros cromáticos

Puesto que ya contábamos con otro mecanismo de evaluación que conseguía algunos resultados, pese a ser éstos notablemente peores que los que en ése momento obteníamos, decidimos reutilizar la funcionalidad de los tableros cromáticos como mecanismo para desempatar entre dos tableros diferentes. Esta era una opción poco costosa de implementar y de probar, puesto que ya disponíamos del código que gestionaba los tableros cromáticos, y consideramos interesante estudiarla.

Los resultados fueron peores de lo esperado. La nueva función de evaluación que empleaba tanto la conectividad como los tableros cromáticos no es que jugase necesariamente peor, pero los resultados no eran demasiado diferentes de lo que ya obteníamos. La mejora que obtuvimos, si es que realmente había alguna, no era lo suficientemente apreciable. Por tanto, se decidió abandonar el concepto de los tableros cromáticos como factor en elegir movimientos.

4.4.3.4 Movilidad y zonas

Ya hemos hablado del concepto de zonas como aquella agrupación de casillas libres que permiten que movamos una bola de una casilla a otra. Pese a ser éste un concepto importante a la hora de detectar los movimientos válidos, hasta este momento no lo habíamos considerado para tomar decisiones.

Relacionado con las zonas, el último concepto importante que vamos a definir en este proyecto, aunque de un modo un poco menos formal que los anteriores, será el de la movilidad.

Entendemos como movilidad para una bola determinada como el número máximo de casillas a las que ésta puede acceder. Pese a estar muy relacionado con el número de zonas, son conceptos diferentes. Si bien el número ideal de zonas será siempre uno (es decir, todas las casillas libres están conectadas con lo cual, cualquier bola que podamos mover podremos

moverla a cualquier casilla libre de la que disponemos), más zonas no implica, necesariamente, un tablero peor en lo que a la movilidad se refiere.

Además, éste concepto nos ayudará en la siguiente situación: suponiendo que tenemos dos bolas igualmente buenas para continuar o finalizar una línea, pero es mejor mover una de ellas si nos permite unir dos zonas en una mucho mayor. En general, consideramos mejor mover las bolas que sirven de frontera entre zonas, o que nos permitan crear zonas mayores, puesto que la movilidad general aumenta para más bolas.

Por ejemplo, dado el siguiente caso:

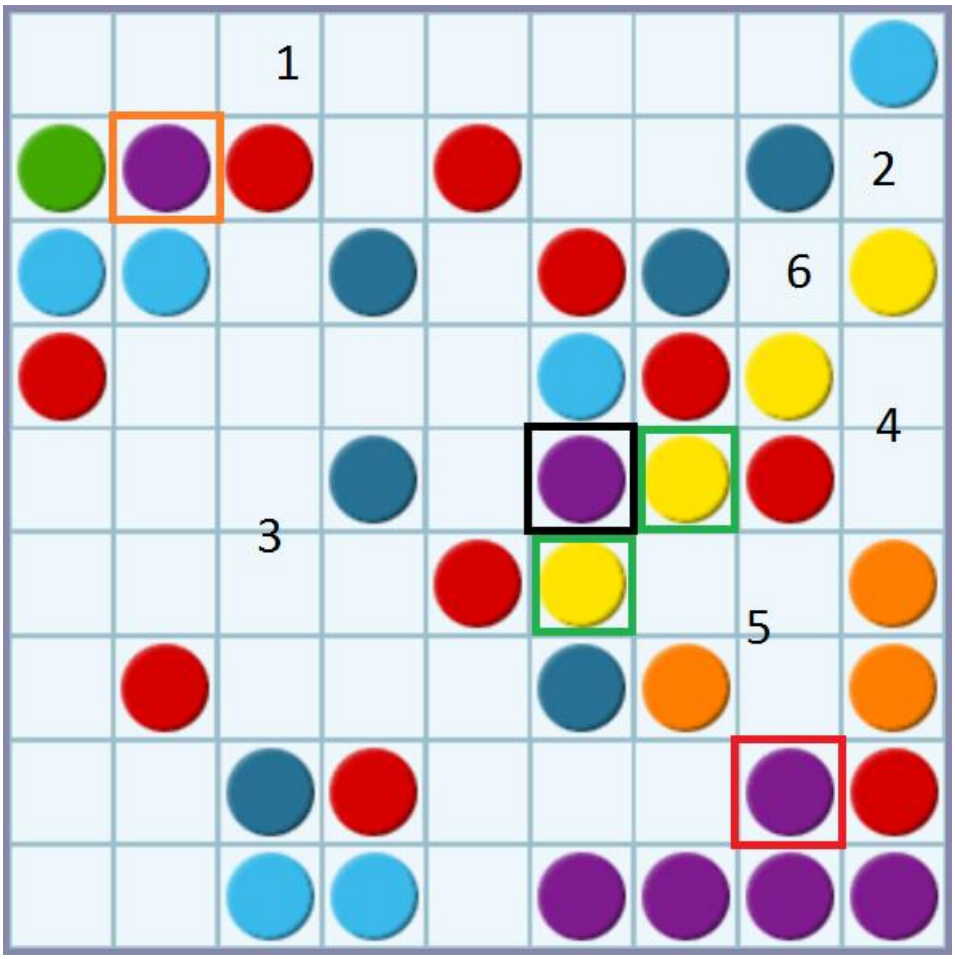

Ilustración 13 – Ejemplo para movilidad y zonas

Para poder realizar línea podemos mover tanto la bola morada marcada en negro como la bola marcada en rojo. Ambas nos permiten ejecutar línea, pero mover la que está marcada en rojo resulta estratégicamente mejor ya que se uniremos las zonas 3 y 5 en una zona considerablemente mayor que moviendo la que está marcada en negra, garantizando acceso a esa nueva zona a todas las bolas que actualmente tienen acceso a la zona 3 y 5. Mientras que la bola marcada en negro sólo permite a las bolas marcadas en verde.

4.4.3.5 Función de evaluación

Tras considerar lo anteriormente descrito decidimos, por lo tanto, crear un mecanismo que nos permitiese potenciar estas ideas como sistema de desempate entre dos tableros

igualmente buenos tras ejecutar sus respectivos movimientos. La idea de este algoritmo es la siguiente:

1. Para dos tableros determinados, investigamos el tamaño de cada una de las zonas en dichos tableros sabiendo, gracias a la conectividad, que llegados a este punto, el número de casillas libres será exactamente el mismo (si necesitamos desempatar, o bien ninguno de los movimientos que estudiamos habrá hecho línea o ambos la habrán hecho).
2. Una vez obtenidos los tamaños de cada una de esas zonas, los ordenamos de mayor a menor.
3. Comparamos los tamaños entre ellos, empezando por el más grande. En el momento en el que dos tamaños no coincidan, decidimos que el movimiento que ha creado un tamaño mayor es mejor.

Tras implementar éste mecanismo de desempate volvimos a ejecutar pruebas con el agente *Greedy* y pudimos observar que la mejoría era lo suficientemente notable como para incluir el análisis de zonas y movilidad en lo que ha resultado ser la función de evaluación.

4.4.4 Agentes

A continuación procederemos a hablar brevemente de los agentes que se han implementado para este PFC.

4.4.4.1 Agente Random

Este agente fue el primero y el más sencillo de implementar. Su estrategia consiste simplemente en calcular todos los movimientos posibles y escoger uno cualquiera al azar de entre ellos.

El objetivo principal de este agente era, por una parte, tener algo contra lo que medir el rendimiento de otros agentes y, por otra, demostrar que mover bolas al azar no es una estrategia válida.

4.4.4.2 Agente Greedy

Éste fue el agente que más tiempo llevó en definir e implementar ya que fue el que se utilizó para refinar la función de evaluación descrita en la sección anterior hasta llegar a la que usamos actualmente.

Si se ha puesto tanto empeño en desarrollar una función de evaluación que permitiese que la estrategia *Greedy* obtuviese los mejores resultados posibles es porque, desde el principio, hemos tenido bastante claro que jugar pensando sólo con uno o dos turnos vista tenía que ser una estrategia muy viable. Parte de éste razonamiento se debe a que todo lo que queda fuera del alcance del jugador es simplemente azar, y éste es tan decisivo que, pasados dos o tres turnos, la cantidad de tableros posibles a los que podemos estar enfrentándonos varían considerablemente.

Finalmente, también necesitábamos obtener la mejor función de evaluación posible para poder continuar desarrollando los demás agentes.

4.4.4.3 Agente UCT Random

Los motivos que nos llevaron tanto a desarrollar este agente como su equivalente utilizando Round Robin fue principalmente la curiosidad. Queríamos saber cómo se comportaría un sistema con algo de inteligencia (el algoritmo UCT con los *X* mejores movimientos preseleccionados por la función de evaluación) escogiendo al azar entre el resto de movimientos para tomar una decisión "informada" de cuál de entre esos *X* movimientos iniciales era el mejor.

La algoritmia aquí es bastante sencilla:

1. En el nodo que representa al estado de la partida actual, filtramos los *X* mejores movimientos para iterar sobre ellos utilizando UCT para determinar cuáles son los más prometedores de entre ellos para explorar uno.
2. Explotamos ese movimiento y generamos azar con la ayuda del simulador (puesto que no tenemos forma de saber dónde caerán las bolas del próximo turno si no hacemos línea).
3. A partir de éste punto, ejecutamos la partida hasta su término, ayudados por el simulador, usando la misma estrategia que el agente aleatorio. Al finalizar, puntuamos el tablero en función de las agrupaciones de "líneas" que se han ido formando, teniendo en cuenta también el número de líneas que se han generado en el camino.

Si bien está claro que los resultados pueden empeorar en comparación con el *Greedy*, en teoría este algoritmo irá mejorando según la partida avanza y el azar en la selección de movimientos se va haciendo un poco menos importante. Además, el agente *Greedy* nos ha demostrado ya la importancia del movimiento actual.

Sin embargo, los resultados obtenidos, pese a ser buenos, no eran del todo satisfactorios. Notamos una bajada considerable en los resultados al final de la partida. Por si esto fuese poco, el agente necesitaba demasiado tiempo para cerrar una sola partida. Esto nos llevó a imponer ciertas restricciones a su implementación *Greedy* como veremos más adelante.

4.4.4.4 Agente Round Robin Random

La motivación para implementar éste algoritmo, junto con la del *Round Robin Greedy* fue ofrecer una alternativa y una comparación frente a los algoritmos que utilizan el algoritmo UCT para escoger que movimientos a explorar. Al contrario que con el UCT, con Round Robin nos aseguramos de que todos los movimientos van a ser explorados la misma cantidad de veces.

4.4.4.5 Agente UCT Greedy

Tras explorar el comportamiento del algoritmo UCT en el caso anterior, decidimos probar a utilizar la función de evaluación para escoger siempre el mejor de los movimientos una vez generado un nodo nuevo. Así pues, implementamos el algoritmo UCT teniendo en cuenta las siguientes desviaciones con respecto al algoritmo.

1. No se guardará en memoria ningún nodo hijo, siempre generaremos uno nuevo con el simulador e iteraremos con él. El motivo que nos llevó a esto es que, salvo muy cerca del final, el factor de ramificación es muy alto. Pese a que pudiésemos guardar todas las combinaciones posibles en el primer nivel, sería muy costoso extenderlo más allá.

Además, más nodos a guardar implica más nodos a explorar. Como la aproximación *Random* necesitaba demasiado tiempo para una sola partida, esto parecía impracticable. Con tantos nodos a explorar, era más que probable que sólo pasásemos por ellos una o dos veces, si queríamos acabar la partida en un tiempo razonable.

2. Por razones similares, no se simulará la partida hasta el final. En primer lugar porque, pese a que la función de evaluación es bastante veloz, es considerablemente más lenta que escoger movimientos al azar. En segundo lugar porque se estimó que pasados dos o tres turnos, las probabilidades de que los tableros se pareciesen a lo que el agente acabaría enfrentándose eran casi despreciables, decayendo exponencialmente cuanta más profundidad explorásemos. Por eso se decidió simular las partidas hasta un máximo de tres turnos.
3. Seguiríamos centrándonos en los mejores X movimientos de acuerdo con la función de evaluación. La experiencia y las pruebas anteriores nos habían demostrado que por cada turno, especialmente a mitad de juego, hay muy pocos movimientos que puedan ser considerados buenos, y que la mayoría de ellos o no aportan nada directamente o son contraproducentes.

A pesar de estas restricciones, el agente sigue necesitando demasiado tiempo para terminar una partida. Si bien, el nivel de juego es muy superior a su versión aleatoria, los resultados obtenidos, cómo se verá en el siguiente capítulo, no distan demasiado de lo obtenido por el agente *Greedy*.

4.4.4.6 Agente Round Robin Greedy

Al igual que su versión aleatoria, ésta se desarrolló para ofrecer un contrapunto con el agente anterior. Los mismos motivos aplican, principalmente ver qué diferencia hay entre explorar los nodos más prometedores más veces frente a explorar todos por igual. El tiempo necesario para finalizar una partida no es muy diferente del que necesita UCT para terminar.

4.5. Conclusiones

En este capítulo hemos mencionado tanto las tecnologías que se han decidido emplear en este desarrollo como la algoritmia que se ha acabado empleando en el mismo, explicando los pasos y razonamientos que nos han llevado hasta la aplicación actual.

Resumiendo, el *Five Lines* ofrece una experiencia de juego completamente configurable para el jugador, y todo el código, tanto del juego como de los agentes, ha sido desarrollado teniendo en cuenta este factor. La implementación se ha realizado además para agilizar algunos procesos que los agentes tendrían que llevar a cabo. Especialmente se ha puesto esfuerzo en asegurar que el algoritmo para determinar qué movimientos son válidos es lo más eficiente posible.

La tarea que más tiempo ha llevado con diferencia, y sobre la cual se ha iterado más, es la función de evaluación. Siendo el azar un factor tan importante en el juego, está bastante claro que jugar sin considerar más allá del turno actual debe ser una estrategia muy válida, aunque quizás no necesariamente la mejor. Es posible que considerar uno o dos turnos más allá ayude a tomar mejores decisiones, basándonos en las probabilidades de que, por ejemplo, aparezca un obstáculo que nos cierre un camino concreto en los próximos turnos.

La función de evaluación que hemos obtenido hace que una estrategia que sólo considera hacer líneas lo antes posible sea lo suficientemente buena.

Los algoritmos más complejos que hemos utilizado requieren demasiado tiempo para finalizar una partida. Mientras que el *Greedy* necesita en torno a un minuto para completar una partida (dependiendo de la cantidad de líneas realizadas), los agentes *Round Robin* y *Greedy* requieren horas. Si bien hay formas de optimizar el código que nos ayuden a reducir considerablemente este tiempo, estos agentes aún necesitan probar que el consumo de tiempo merece la pena. A pesar de todo, no descartamos que con un código más optimizado y unos mecanismos de evaluación un poco más afinados, estos agentes, aparte de ser más rápidos brindarán resultados notablemente mejores que los de la estrategia *Greedy*.

Capítulo 5
Resultados

En éste capítulo revisaremos los resultados obtenidos al finalizar el desarrollo del proyecto. Nos concentraremos, fundamentalmente, en analizar los diferentes agentes después de la ejecución de cien partidas con cada uno de ellos (a excepción del agente *Random*, para el cual se decidió ejecutar mil partidas), hablando también del tiempo que lleva a cada uno finalizar una partida y el resultado obtenido en ésta, utilizando una gráfica para representar el resultado en líneas de las partidas. Una vez analizado cada agente, tendremos una pequeña sección dedicada a compararlos entre ellos.

5.1. Resultados en función del número de líneas por agente

A continuación mostraremos los resultados de los agentes tras ejecutar el set de partidas propuesto para cada uno. Antes de pasar a analizar cada uno por separado, aquí hay una breve tabla resumen.

Agente	Promedio de líneas
Random	0.042
Greedy	72,41
UCT Random (Filtro = 3)	49,54
UCT Random (Filtro = 2)	56,8
Round Robin Random (Filtro = 3)	48,58
Round Robin Random (Filtro = 3)	59,3
UCT Greedy (Filtro = 2)	74,1
Round Robin Greedy (Filtro = 2)	69,9

Tabla 20 - Promedio de líneas por agente

5.1.1 Agente Random

Tras ejecutar mil (1000) partidas con él, lo cual le lleva aproximadamente medio minuto, el agente *Random* ha conseguido puntuar sólo en cuarenta de ellas, y sólo ha conseguido hacer más de una línea en dos de esas partidas.

Con esto podemos asegurar que realizar movimientos al azar no solamente es una estrategia muy mala, sino que además es casi seguro que jugando así no se obtendrá ningún resultado al finalizar la partida.

Para este agente no se muestran ningún tipo de gráficas por no considerarse necesario, ni se volverá a mencionar en los siguientes estudios. El promedio de líneas por partida obtenido es de sólo 0,042.

5.1.2 Agente Greedy

En términos del tiempo que tarda en resolver una partida y los resultados obtenidos, este agente es el que mejores resultados ofrece. Termina de ejecutar el lote de cien (100) partidas en aproximadamente dos horas, lo cual implica poco más de un minuto por cada partida. Además, como se verá más adelante, sólo hemos obtenido resultados algo mejores con el algoritmo *UCT Greedy*, pero necesitando considerablemente más tiempo para ello.

En el lote de pruebas ejecutado para este estudio, estos son algunos datos de interés, utilizando, como hemos mencionado, el número de líneas realizado por partida:

Concepto	Valor
Media	72,41
Desviación Estándar	41,671
Máximo	242
Mínimo	15
Primer Cuartil	42,25
Mediana	61
Tercer Cuartil	87,75

Tabla 21 – Estadísticas del agente *Greedy*

La distribución gráfica de los resultados, ordenados de mayor a menor, quedaría así:

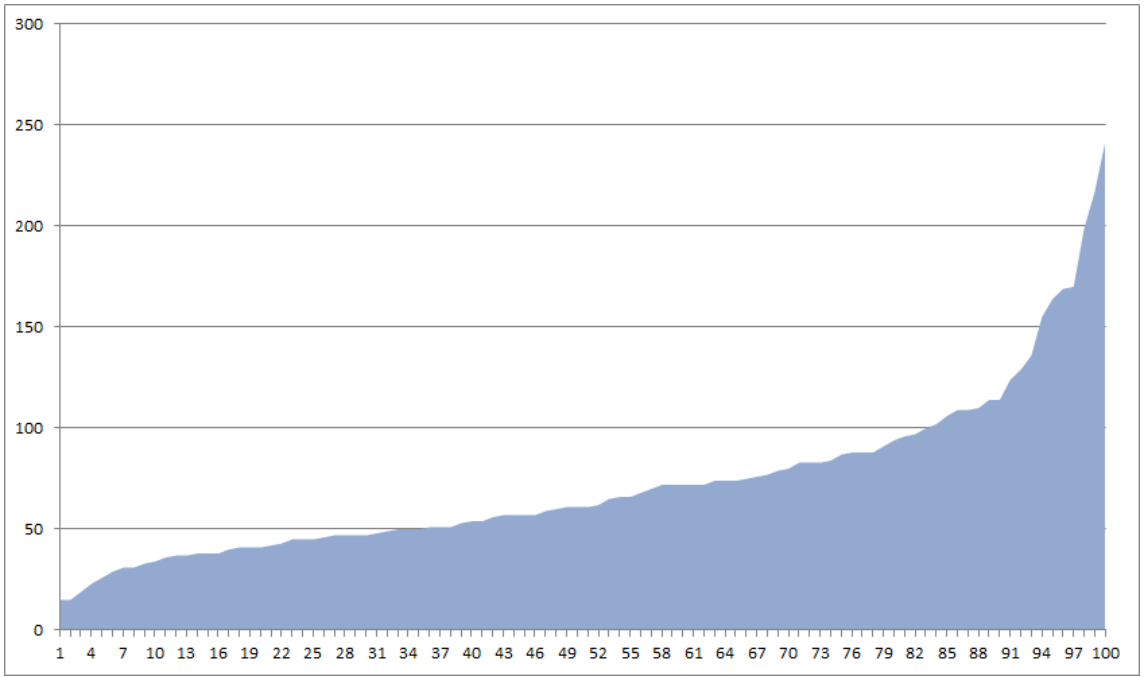

Ilustración 14 – Distribución gráfica de resultados del agente *Greedy*

Los resultados obtenidos son, por lo tanto, bastante satisfactorios, si bien pueden estar algo lejos de lo que un humano profesional puede llegar a conseguir en la mejor de sus partidas. Pese a todo, esto confirma que hay estrategias válidas a la hora de jugar el juego, y que jugar sin tener demasiada consideración por los turnos venideros (más allá de considerar qué bolas entrarán en la partida el próximo turno) es una buena estrategia.

5.1.3 Agente UCT Random

Los resultados obtenidos por este agente varían en gran medida, como era de esperar, en el filtro aplicado previamente por la función de evaluación, preseleccionando los mejores movimientos del estado actual antes de proceder a iterar sobre ellos. Una vez identificados estos movimientos, el agente ejecutará una partida hasta el final seleccionando movimientos al azar hasta finalizarla. Como se comentó en el capítulo anterior, el tablero resultante de esta partida se analiza, premiando la formación de líneas parciales sobre el mismo, y teniendo también en cuenta el que se haya realizado alguna línea completa antes de llegar al final. Pese a que los resultados obtenidos palidecen frente a los obtenidos por el agente *Greedy* distan de ser realmente malos.

Los resultados obtenidos para una implementación UCT Random que explora los tres mejores movimientos según la función de evaluación queda como sigue:

Concepto	Valor
Media	49,54
Desviación Estándar	24,581
Máximo	161
Mínimo	15
Primer Cuartil	33,25
Mediana	44,5

Tercer Cuartil	56,75

Tabla 22 – Estadísticas del agente *UCTR-3*

Mientras que la distribución gráfica de los resultados queda:

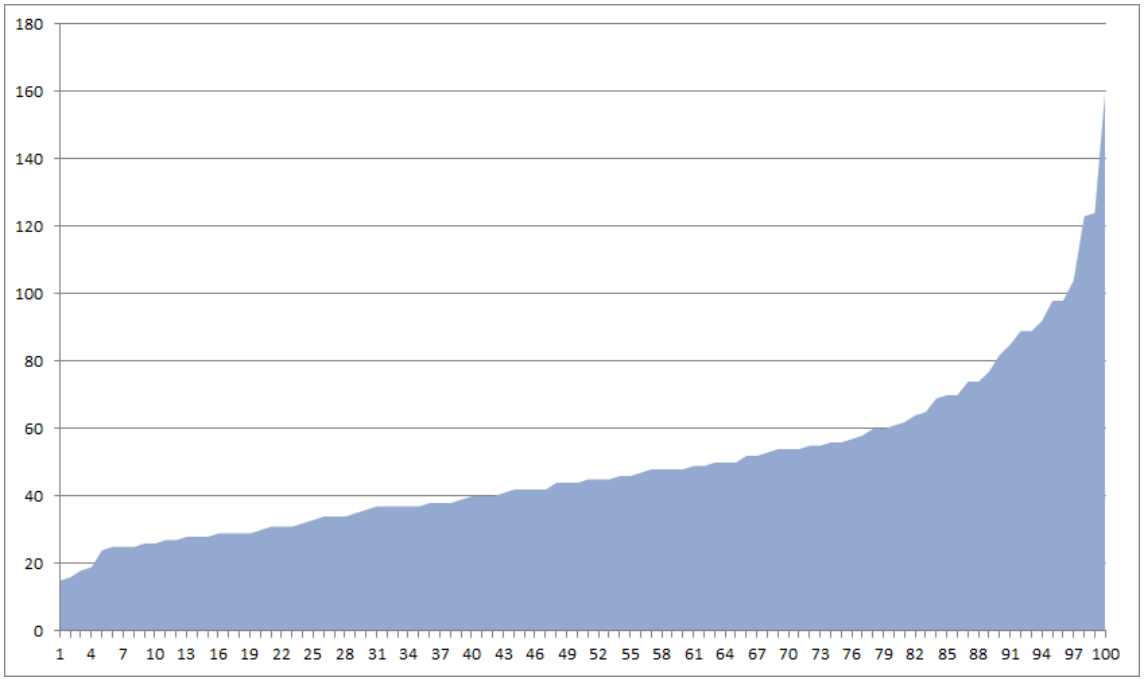

Ilustración 15 – Distribución gráfica de resultados del agente *UCTR-3*

Se puede apreciar que éstos resultados están muy lejos de los obtenidos por el agente *Greedy*, pese a ser razonablemente decentes, y representan una caída de entre un 30 y un 35% aproximadamente en la media de líneas por partida. Los resultados buenos para este agente son considerablemente peores que los obtenidos por el *Greedy*, y los datos tienen una distribución considerablemente más baja.

Tras ese lote de pruebas, se ejecutó otro lote filtrando sólo los dos mejores movimientos para ver el impacto que esto podría tener:

Concepto	**Valor**
Media	56,8
Desviación Estándar	24,408
Máximo	171
Mínimo	22
Primer Cuartil	39,25
Mediana	53
Tercer Cuartil	69,75

Tabla 23 – Estadísticas del agente *UCTR-2*

Con la siguiente representación gráfica.

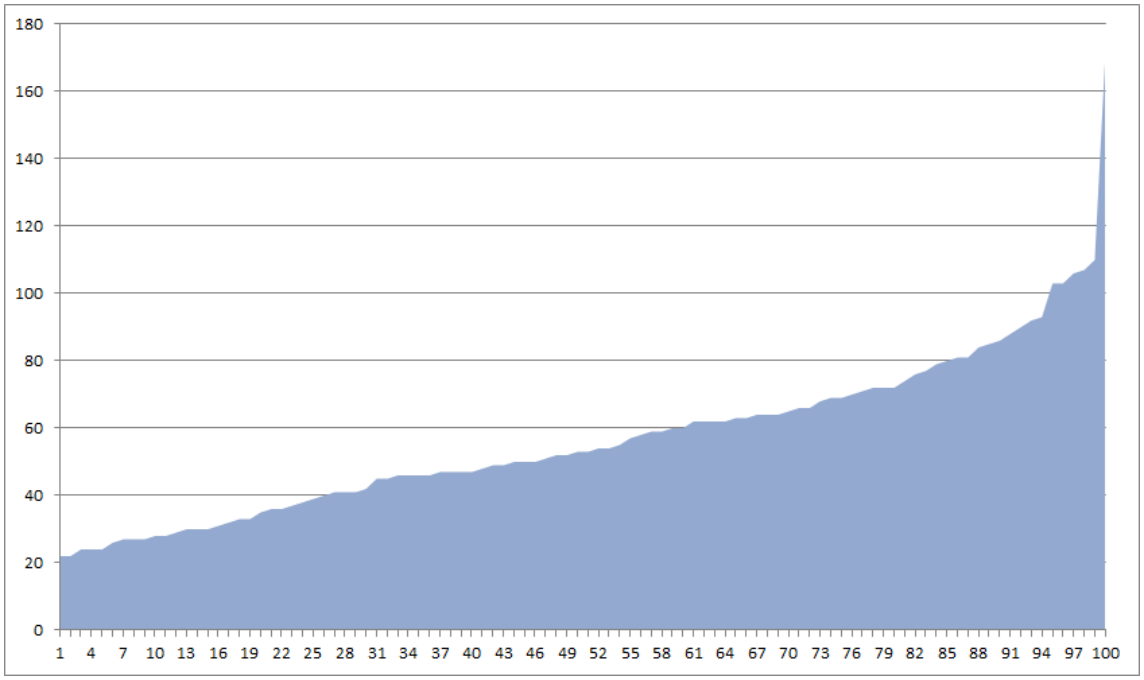

Ilustración 16 – Distribución gráfica de resultados del agente *UCTR-2*

La mejora tras reducir el número de movimientos a explorar es menor, puesto que estamos limitando un poco el efecto que el azar tiene sobre escoger los siguientes movimientos al azar y realizar movimientos sobre ellos, con una caída con respecto al agente *Greedy* más cercana al 25%. Las partidas realmente "buenas" distan bastante de las mejores del *Greedy*, pero la distribución de resultados es considerablemente mejor que las del caso anterior.

No se realizaron pruebas con un filtro menor, puesto que esto implicaría que jugaría exactamente como el agente *Greedy*, aunque necesitando más tiempo.

5.1.4 Agente Round Robin Random

Al igual que con los agentes anteriores, la implementación *Round Robin Random* depende considerablemente de la cantidad de movimientos empleada como filtro. Se repitieron las mismas pruebas, es decir, aplicando un filtro de tres y luego uno de dos.

Los resultados para el primer caso de pruebas son:

Concepto	Valor
Media	48,58
Desviación Estándar	20,204
Máximo	110
Mínimo	13
Primer Cuartil	32
Mediana	44
Tercer Cuartil	63,5

Tabla 24 – Estadísticas del agente *RRR-3*

Y su distribución gráfica es:

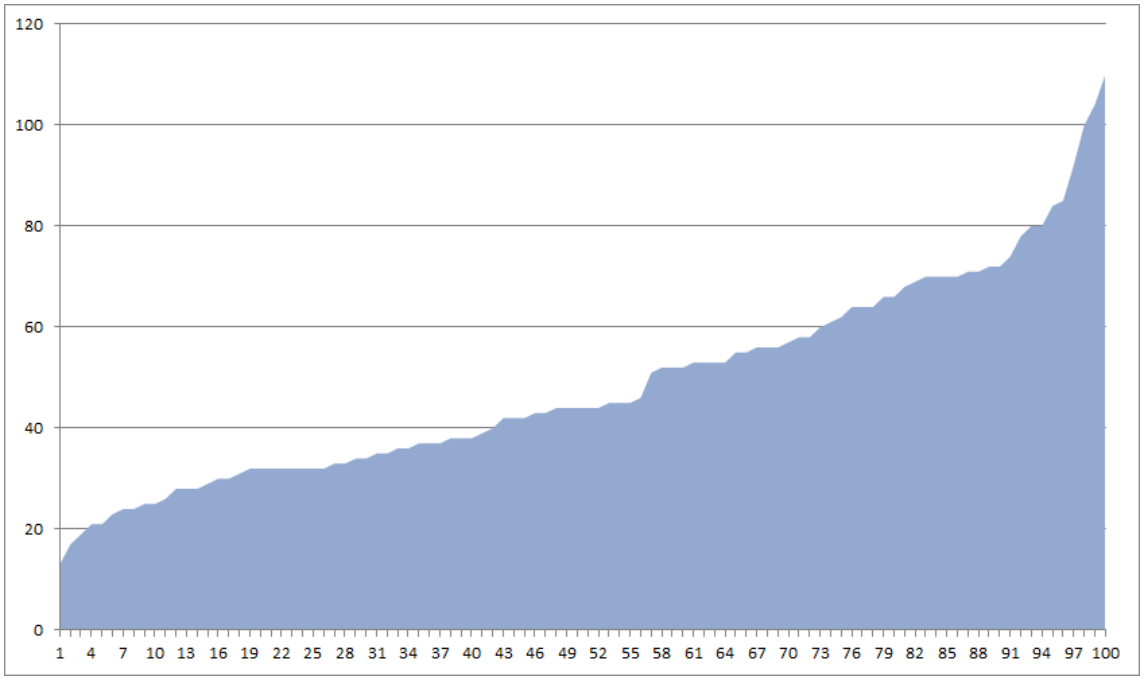

Ilustración 17 – Distribución gráfica de resultados del agente *RRR-3*

Cabe destacar que los resultados obtenidos por éste agente son algo mejores que los obtenidos por su equivalente en *UCT*, exceptuando la media y los datos atípicos. Pese a todo, las diferencias entre ambos no resultan lo suficientemente notables.

El segundo lote de pruebas, en el cual empleamos un filtro de dos movimientos, nos brinda los siguientes resultados:

Concepto	**Valor**
Media	59,3
Desviación Estándar	26,37
Máximo	190
Mínimo	16
Primer Cuartil	41
Mediana	54
Tercer Cuartil	70,5

Tabla 25 – Estadísticas del agente *RRR-2*

Siendo su distribución gráfica la siguiente:

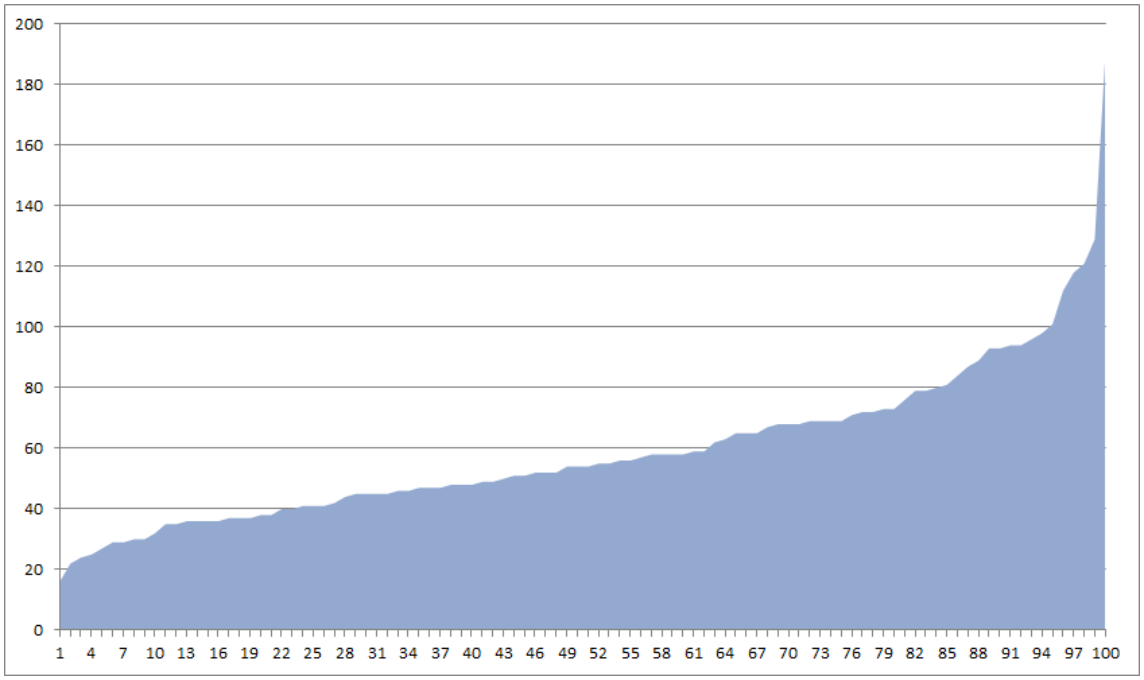

Ilustración 18 – Distribución gráfica de resultados del agente *RRR-2*

Al igual que en el caso anterior, los resultados obtenidos parecen ligeramente mejores que los obtenidos por su equivalente UCT. La media es notablemente mayor, y los datos parecen favorecer posiciones algo más altas en general. No obstante la diferencia entre ellos no es la suficiente como para poder afirmar cuál de las dos estrategias es más válida.

5.1.5 Agente UCT Greedy

Debido a los resultados obtenidos por las pruebas anteriores, y dado que la recopilación de datos de prueba es muy costosa en tiempo, decidimos realizar sólo un lote de pruebas tanto con éste agente como con el siguiente. Los tiempos de ejecución de ambos se dispararon aún más con éstos, necesitando del orden de un día para finalizar tan sólo en torno a cinco partidas. Por ejemplo, para una partida en la que se lograron 114 líneas, el agente *UCT Greedy* necesitó de unas siete horas para finalizarla.

Siendo éste el caso, se decidió aplicar un filtro de dos movimientos a ambos agentes para tener datos para comparar entre ellos y el *Greedy*. En los dos casos, nunca se fue a una profundidad más allá de tres movimientos en el futuro, ya que se consideró que la partida resultante dejaba de ser representativa más allá.

Tras el lote de pruebas de cien movimientos los resultados obtenidos fueron los siguientes:

Concepto	Valor
Media	74,41
Desviación Estándar	39,94
Máximo	184
Mínimo	22
Primer Cuartil	48,25

Mediana	63,5
Tercer Cuartil	94

Tabla 26 – Estadísticas del agente *UCTG*

La distribución gráfica de estos resultados es la que sigue:

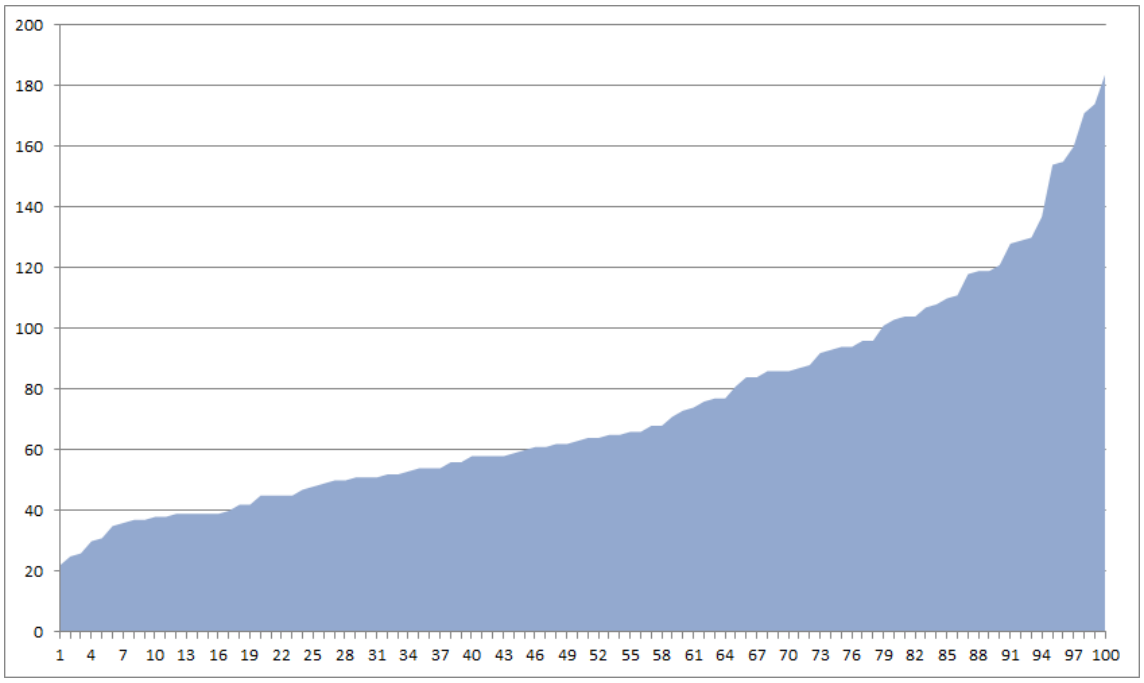

Ilustración 19 – Distribución gráfica de resultados del agente *UCTG*

Como se puede observar, a excepción de que los datos atípicos que elevan la media no alcanzan los niveles a los que el *Greedy* llega, los resultados obtenidos por éste agente son razonablemente superiores a los del primero. Sus partidas consiguen anotar más frecuentemente más de cien líneas y baja de cincuenta con menos frecuencia. El mayor inconveniente que éste agente tiene con respecto al *Greedy* es que el tiempo de ejecución es mucho mayor, y sus resultados, pese a ser mejores, pueden no justificar el tiempo.

5.1.6 Agente Round Robin Greedy

La última implementación por comentar es la del agente *Round Robin Greedy*, para la cual, al igual que con la anterior, se estableció un filtro de dos movimientos y tres turnos de profundidad. También nos aseguramos de que el número de exploraciones fuese equivalente en ambos casos (estableciendo 60 exploraciones como máximo en ambos, igualmente distribuidas para el *Round Robin*). Como nota sobre su rendimiento, en una partida de ejemplo en la que éste agente consiguió 108 líneas tardó en finalizarse casi ocho horas.

Los resultados obtenidos son los siguientes:

Concepto	Valor
Media	69,97
Desviación Estándar	31,38
Máximo	182
Mínimo	28

Primer Cuartil	46,75
Mediana	62
Tercer Cuartil	83,75

Tabla 27 – Estadísticas del agente *RRG*

Y su representación gráfica es la que sigue:

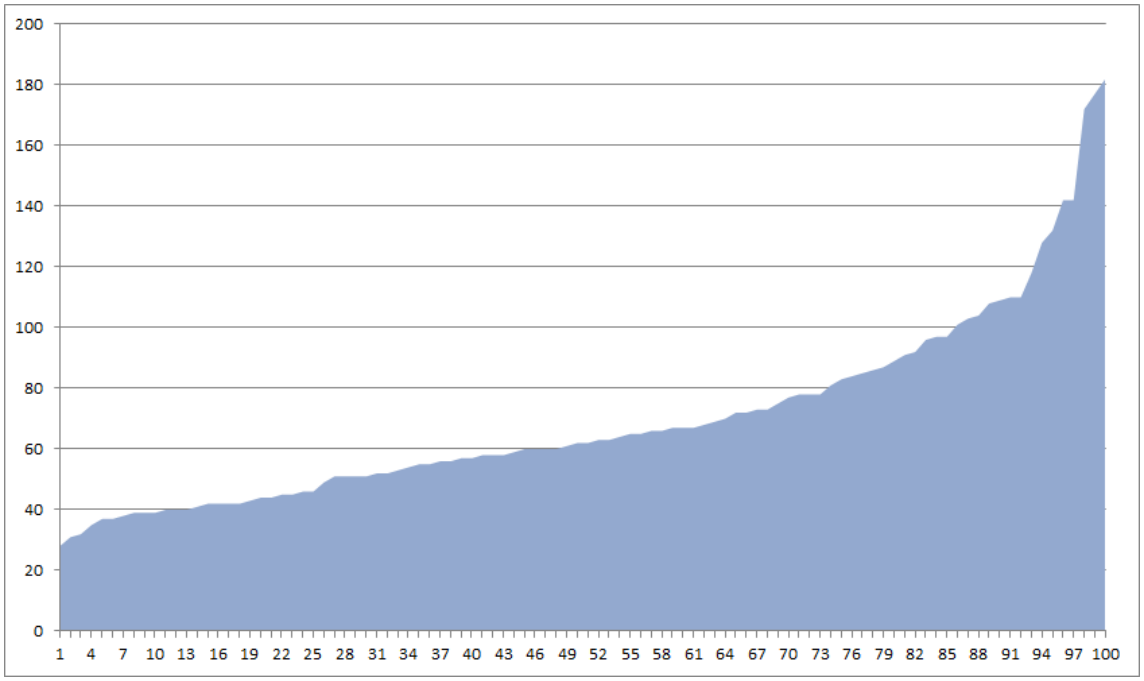

Tabla 28 – Distribución gráfica de resultados del agente *RRG*

En éste caso, los resultados son algo más similares que los del *Greedy*, incluso se podría argumentar que mejores. Las máximas alcanzadas son menos explosivas, pero las mínimas son considerablemente mejores. La media es más baja, pero también el número de datos atípicos que pueden impulsarlas es menor. Comparándolo con respecto a su equivalente *UCT*, los resultados son algo peores y requiere algo más de tiempo en obtenerlos, como hemos comentado anteriormente.

5.2 Comparación entre agentes

Para finalizar éste estudio, presentaremos una pequeña tabla con los resultados obtenidos por cada uno de los agentes, marcando en verde el mejor resultado y en rojo el peor:

	Greedy	UCTR 3	UCTR 2	RRR 3	RRR 2	UCTG	RRG
Media	72,41	49,54	56,8	**48,58**	59,3	**74,41**	69,97
Desviación	41,671	24,581	24,408	20,204	26,37	39,94	31,38
Máximo	**242**	161	171	**110**	190	184	182
Mínimo	15	15	22	**13**	16	22	**28**
Cuartil 1	42,25	33,25	39,25	32	41	**48,25**	46,75
Mediana	61	44,5	53	**44**	54	**63,5**	62
Cuartil 3	87,75	**56,75**	69,75	63,5	70,5	**94**	83,75

Tabla 29 – Comparación de las estadísticas entre los agentes

Y, la comparación gráfica es la siguiente:

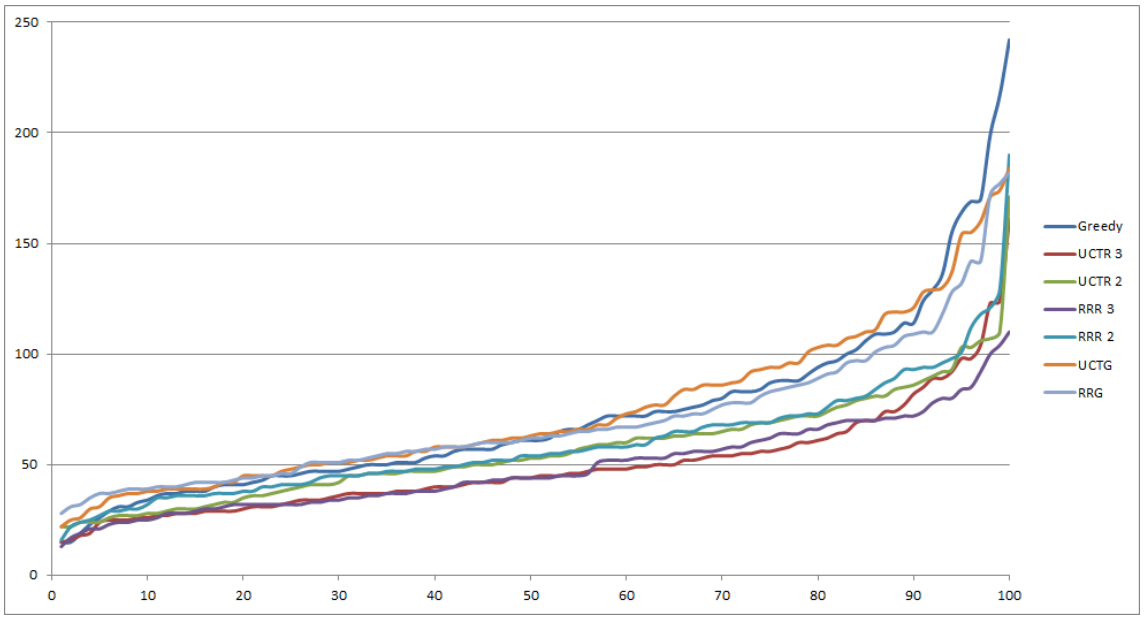

Ilustración 20 – Comparación gráfica de los resultados de los agentes

Se puede apreciar con ésta gráfica que los tres agentes mejor posicionados son los tres *Greedys*. El *Greedy* normal tiene resultados algo peores que los otros dos para la primera mitad de los datos obtenidos, pero considerablemente mejores para el último 10% de ellos. *Round Robin Greedy* (RRG en la tabla) tiene resultados parecidos a los del *Greedy* hasta llegar al ochenta por ciento de los datos. *UCT Greedy* (UCTG en la tabla) por su parte es consistentemente mejor que el *Greedy* exceptuando los resultados más elevados, y está a la par que *Round Robin* en los resultados más bajos.

5.3 Conclusiones

Los resultados obtenidos nos indican que el agente que mejores resultados obtiene y el más consistente a lo largo de su ejecución es el agente *UCT Greedy*. Es interesante destacar que este agente debe decidir entre dos movimientos, tomando, en ocasiones, el que la función de evaluación dicta que es peor. Es cierto que las mejoras obtenidas no son tan significativas como para poder afirmarlo con una seguridad total, y aún menos para justificar el tiempo tan significativamente elevado que éste requiere frente a la aproximación *Greedy*.

Sin embargo, como se ha mencionado anteriormente, la aplicación aún puede mejorarse tanto en términos de eficiencia que podrían reducir este tiempo de ejecución considerablemente, como en términos de las valoraciones extra que los agentes más complejos emplean para decidir cómo fue el resultado final de la simulación.

Capítulo 6
Líneas Futuras

En éste capítulo revisaremos qué mejoras se podría aplicar a la aplicación para hacerla más completa de lo que esta versión es.

6.1 Optimizar eficiencia del código para los agentes

Pese a que el código del juego es lo suficientemente rápido para que un jugador humano perciba sus comandos ejecutándose "en tiempo real", el agente *Random* necesita sólo unos segundos para finalizar una partida y el agente *Greedy* puede finalizar una partida razonablemente larga, del orden de unas cien líneas, en unos pocos minutos, los demás agentes tardan horas en completar una partida corta (30 líneas).

Para mejorar esto, podría bastar con optimizar las tareas de limpieza y comprobación ejecutadas en la clase Tablero y derivadas, utilizar estructuras de datos más veloces y cambiar algunos cálculos.

6.2 Añadir otros algoritmos de I.A.

Podría ser interesante probar con otros algoritmos de Inteligencia Artificial y probar qué resultados obtienen contrastados con los agentes actualmente estudiados.

6.3 Mejorar los mecanismos de evaluación de estados

Actualmente hay dos mecanismos de evaluación de estados:

- La función de evaluación.
- La evaluación del tablero al finalizar la simulación (sea por turnos sea hasta el final de la partida).

Estos mecanismos actualmente dan el mismo valor a combinaciones que no son, en absoluto equivalentes. Sin entrar en demasiado detalle y sin complicar en exceso el código (lo cual entraría en conflicto con el primer punto de este apartado), habría que no suponer que una línea de tres no tiene el mismo valor esté en la zona del tablero en la que esté, por ejemplo, centro vs una diagonal de longitud máxima 4 en una esquina. Cuanto impacto tendría esto en los resultados podría ser interesante de medir.

6.4 Hacer el juego más atractivo para jugadores humanos

El juego en su versión actual permite a un jugador humano jugar con él. Sin embargo, puesto que el objetivo del proyecto nunca fue crear un juego, si no evaluarlo desde el punto de vista de la Inteligencia Artificial, actualmente sólo dispone de una interfaz en modo texto, con los colores representados como números, y control por teclado introduciendo coordenadas de origen y destino. Estos dos factores hacen que no sea muy agradable jugar a la versión actual.

Para solucionar esto, hay dos cambios fundamentales que hacer:

- Incluir una interfaz gráfica similar a las versiones ya existentes del juego.
- Implementar controles *point-and-click*.
- Sonido y animaciones.

6.5 Sistema de ayudas para humanos

Se podría implementar un sistema de ayuda utilizando alguno de los agentes ya implementados. El jugador podría, dada una situación particular del tablero, solicitar al agente que le marque el movimiento más recomendable. El tiempo que éste agente deberá tomar para sugerir un movimiento deberá ser casi imperceptible para un jugador humano, por ejemplo, no más de 1 ó 2 segundos.

6.6 Ampliar el juego incluyendo más reglas

Actualmente, el *Five Lines* sólo considera matrices rectangulares *MxN* perfectas, sin huecos dentro del tablero de juego. Un posible cambio sería la implementación de éstos huecos e incluso permitir otro tipo de figuras como tablero de juego. Esto tendrá un poco de impacto en los mecanismos que los agentes utilizan para tomar decisiones, pero éste debería ser mínimo.

6.7 Comprobar las características de una partida antes de iniciarla

Actualmente, el código del *Five Lines* genera partidas para cualquier combinación de elementos posibles (dimensión del tablero, longitud de línea, colores…). Al no entrar en los límites del proyecto se ha desestimado estudiar estos elementos antes de iniciar la partida, con lo cual ciertas combinaciones de elementos harán partidas imposibles o partidas no

interesantes. Por ejemplo, hacer un tablero de demasiada poco tamaño hará que hacer una sola línea sea imposible; tener demasiados pocos colores podría llevar a que la partida no pudiese terminar nunca…

6.8 Utilizar ficheros de configuración para las características de los agentes

Otro elemento con el que se podría mejorar el proyecto actual y que, por tiempo y por entrar fuera de los límites originales del proyecto (i.e.: no impacta en el estudio de los agentes en sí), no se ha realizado es el implementar un mecanismo para que, a partir de un fichero de configuración, poder modificar algunas de las características de los agentes (excluyendo Random y Greedy) sin necesidad de actualizar el código. Estos elementos serían, por ejemplo:

- Número de turnos a jugar durante las partidas simuladas, cuando aplique.
- Número de veces a iterar dentro de un mismo turno.
- Valor de *épsilon* para UCT.

6.9 Crear una función de evaluación que considere también el histórico de la partida

En el código actual, disponemos de un pequeño modulo que va recopilando información sobre cuantas bolas de cada color han ido apareciendo hasta el momento. Ésta información sólo se utiliza para imprimirla al final de cada partida si se solicita esta información. Sería interesante generar una función de evaluación que emplee éstos datos para intentar predecir qué colores tienen más probabilidades de aparecer en los turnos venideros e intentar actuar en consecuencia.

Capítulo 7
Conclusiones

En éste capítulo revisaremos los objetivos planteados en este proyecto y veremos en qué medida se han alcanzado éstos. También hablaremos un poco sobre los aspectos menos satisfactorios del mismo.

7.1 Revisión de objetivos

A continuación miraremos los objetivos planteados en el comienzo del proyecto y veremos en qué medida hemos cumplido cada uno de ellos.

7.1.1 Five Lines

Hemos realizado una implementación exitosa del juego planteado. Todos los elementos del juego son completamente parametrizables con lo que se puede jugar y realizar estudios para cualquier combinación de elementos.

7.1.2 Separación entre juego y agente

Pese a estar dentro del mismo paquete ejecutable, el juego y los agentes son “independientes” entre sí. Es decir, el agente en ningún momento dispone de más información de la que dispondría un jugador humano ya que las colocación de las bolas extra de un turno no se decide hasta después de que el juego haya recibido un movimiento, y no se deciden las siguientes bolas hasta después de colocar las nuevas. Tanto las posiciones como como los colores de las bolas del próximo turno se generan aleatoriamente en el momento en el que se necesitan.

Otra medida tomada para asegurar la independencia es limitar los puntos de comunicación entre ambos. Lo único que el Agente envía a la partida es un movimiento, y lo único que el juego envía al agente es una copia del tablero de la partida para asegurarnos de que el tablero real queda intacto.

7.1.3 Función de evaluación

La función de evaluación empleada es bastante satisfactoria. Prueba de ello es que el agente *Greedy* es bastante efectivo y eficiente. Es bastante rápida en evaluar los tableros resultantes tras cada movimiento (menos de un milisegundo por cada uno) y nos ofrece resultados fiables que nos permiten decidir con entre dos estados diferentes.

7.1.4 Agentes

Ahora dedicaremos un par de líneas a cada uno de los agentes implementados para este estudio:

- *Random*: este agente nos demuestra que en el *Five Lines* realizar movimientos al azar no es en ningún caso una estrategia viable. Casi todas sus partidas acaban sin realizar ninguna línea.
- *Greedy*: este agente es, actualmente, el que mejores resultados ofrece. Utilizando sólo la función de evaluación para escoger el mejor movimiento de los posibles, tiene puntuaciones buenas tardando muy poco tiempo en finalizar una partida.
- *UCT/Round Robin Random*: estos agentes prueban una vez más que, incluso metiendo algo de algoritmia, tener una buena función de evaluación es vital. Tras aplicar un filtro inicial a los mejores *X* movimientos, los resultados mejoran, volviéndose mejores cuanto menor es *X*. Sin embargo, el mover al azar tras el filtro del primer movimiento, hace que éstos resultados no sean lo suficientemente satisfactorios.
- *Round Robin Greedy*: a pesar de que este agente obtuvo resultados notables, no consiguió llegar a los obtenidos por el *Greedy*, quedando notablemente detrás y requiriendo mucho más tiempo de ejecución.
- *UCT Greedy*: finalmente, este agente obtiene resultados lo suficientemente próximos al *Greedy* como para no notarse demasiada diferencia con él, exceptuando el tiempo que necesita para tomar decisiones. Y eso escogiendo entre más movimientos. Puede que mejorando la evaluación de estados finales y optimizando el código éste pudiese acabar siendo el agente que mejores resultados ofreciese.

El elemento que más claro queda de este estudio es que el *Five Lines* es un juego en el cual, al ser el azar un elemento tan importante, elegir jugadas sin considerar más allá de un turno vista (es decir, sólo con la información de la que se dispone en el momento) es una estrategia viable aunque probablemente no la mejor.

7.2 Limitaciones

Ahora hablaremos de los puntos menos satisfactorios identificados al finalizar este proyecto.

7.2.1 Rendimiento en los agentes

Todos los agentes que emplean un algoritmo de Inteligencia artificial son mucho más lentos y menos efectivos de lo esperado. De todos los seleccionados, sólo uno, *UCT Greedy*, juega al mismo nivel que el agente Greedy (*Round Robin Greedy*, está cerca, pero no lo suficiente). Y, mientras que el agente Greedy sólo necesita en torno a dos horas para terminar 100 partidas, los otros tardan del orden de días en finalizar para darnos resultados un poco peores.

Si bien esto podría interpretarse como que una aproximación directa al juego (es decir, no actuar pensando más allá del turno actual) es la mejor, no podemos realmente plantear esa conclusión. Hay bastantes factores que afectan a ambos elementos (tiempo y resultados) que deberíamos mejorar antes de afirmar eso. El código actual no es todo lo eficiente que podría ser, lo cual impacta mucho más a estos agentes, y muy posiblemente las evaluaciones que realizamos para los nodos hoja obtenidos (i.e.: valoración del tablero resultante) no son las mejores posibles.

Capítulo 8
Planificación y Presupuesto

En este capítulo analizaremos el proceso de planificación del presente proyecto, y haremos un estudio de cuán fiel ha sido el desarrollo real frente a lo estimado. También se evaluará los costes en los que se han incurrido frente a los costes previstos.

8.1. Planificación

El proyecto se dividió fundamentalmente en tres tareas, cada una con sub tareas. El listado es, por tanto, el siguiente:

1. El juego *Five Lines*: Implementación del juego que utilizaremos. Las subfases serían:
    a. Estudio de las reglas y estados del juego.
    b. Implementación de una versión del juego que cumpla nuestros requisitos, en especial poder configurar una partida con cualquier combinación de elementos.
    c. Probar el juego.
2. Función de evaluación: análisis del juego para obtener una función de evaluación que permita distinguir, entre dos tableros cualesquiera, cual es en general más preferible.
3. Agentes: Implementación de los agentes descritos en el Capítulo 3:
    a. *Random*: elige un movimiento cualquiera y lo ejecuta
    b. *Greedy*: escoge siempre el mejor movimiento dictado por la función de evaluación. Su implementación va en paralelo a la de la función de evaluación.
    c. *Montecarlo UCT*: en sus dos variedades, *Greedy* y *Random*. Previo filtro de los mejores X movimientos según la función de evaluación.
    d. *Round Robin*: Igual que el caso anterior.
4. Pruebas y análisis de resultados: resultados que cada agente consigue tras ejecutarse cien partidas con ellos.
5. Documentación del proyecto.

8.1.1 Planificación original

A continuación se ofrece un diagrama de Gantt con la planificación de las fases anteriormente comentadas. Al contar con sólo una máquina para realizar pruebas, se decidió, por evitar sobrecargarla y meter ruido en los resultados, especialmente los tiempos necesarios para que cada agente necesita para terminar su lote de pruebas, no ejecutar las pruebas concurrentemente, si no esperar a que terminase el actual antes de comenzar con el siguiente.

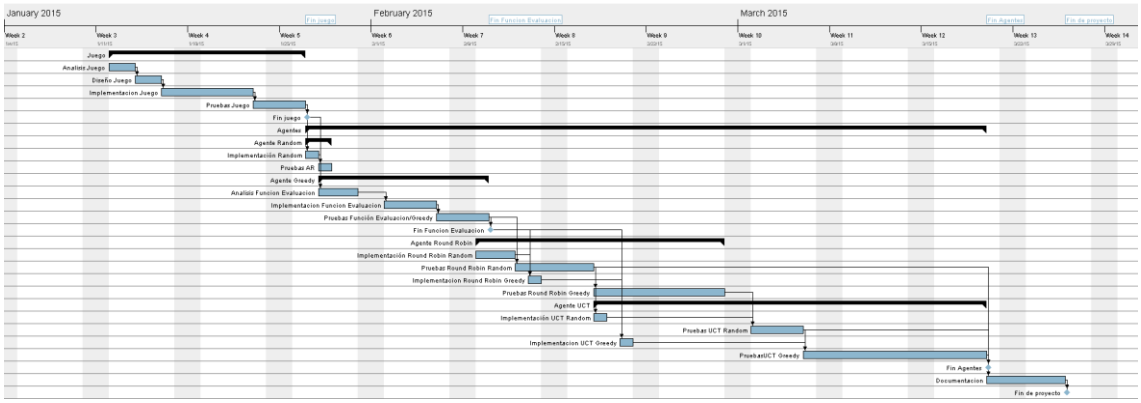

Ilustración 21 – Planificación original

8.1.2 Desarrollo real

Debido a varios imprevistos el proyecto no se realizó de forma continuada, habiendo una pausa notable tras la implementación y pruebas del algoritmo *Greedy*. Hubo otros factores que afectaron notablemente al desarrollo, particularmente:

- Alcanzar una función de evaluación lo suficientemente buena nos llevó bastante más tiempo del planificado. Esto fue causado por las numerosas iteraciones por las que pasamos antes de encontrar una solución que podíamos considerar como la solución final.
- Las pruebas de los agentes *Round Robin* y *UCT* llevaron muchísimo más tiempo del que se había estimado en un principio. Al no disponer de dos máquinas idénticas, no fue posible ejecutar las pruebas en paralelo. Pese a que el tiempo de pruebas cuadruplicó el tiempo planificado, éste tiempo no fue del todo muerto, ya que se empleó para otras actividades ajenas al proyecto.

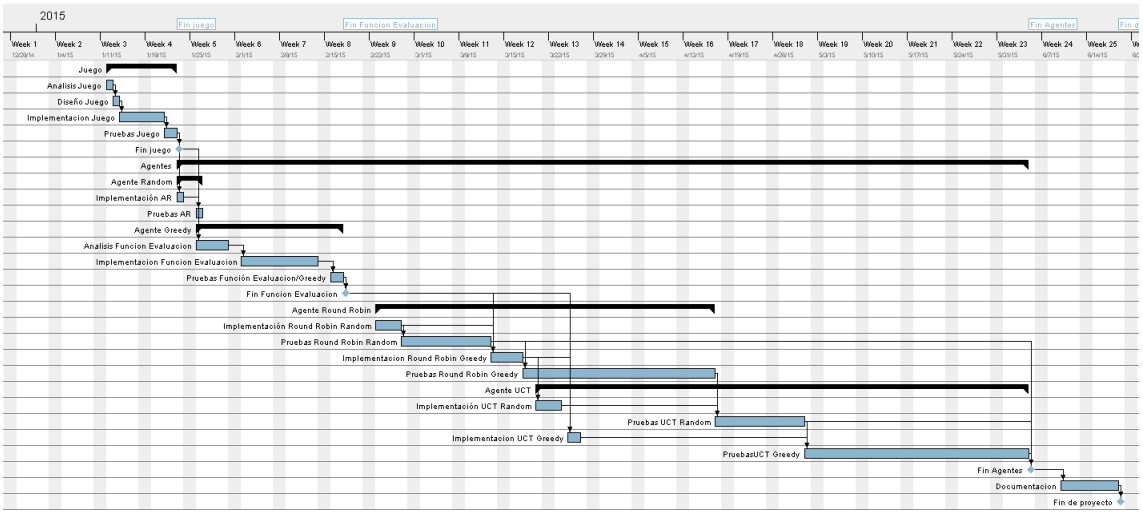

Ilustración 22 – Desarrollo real

Como se puede observar, el proyecto sufrió un retraso de más de diez semanas desde su inicio hasta su finalización, si bien no todo éste tiempo puede considerarse perdido, al asignar parte de los recursos del proyecto a otras tareas.

8.2. Recursos

En esta sección detallaremos todos los recursos (Hardware, Software y Humanos) empleados durante el desarrollo de este proyecto. También se indicará a su lado el coste que cada uno de éstos para poder evaluar más adelante los costes del proyecto.

Ya que siempre se ha buscado trabajar con herramientas *Open Source* o gratuitas cuando había la opción, el coste de muchas de ellas es de 0€. Además, tanto los costes de *Microsoft Office* como de *Windows 7 Professional* vienen incluidos en el coste de la máquina de desarrollo.

Sección	Recurso	Precio
Hardware	Equipo de desarrollo	1.100 €
	Máquina de Backup/servidor SVN	630€
Software	Windows 7 Professional	0€
	SLES 11 (SUSE Linux Enterprise Server)	0€
	Microsoft Office 2010	0€
	NetBeans IDE 7.0	0€
	SVN	0€
	Tortoise SVN	0€
	Dia	0€
	Ganttproject	0€
	Altova UModel (Trial)	0€
Humanos	Salario bruto analista/programador	24.000€ / año
	Costes adicionales empleado	7.200€ / año

Tabla 30 - Recursos

Nota: Para los costes adicionales del empleado sólo se han tenido en cuenta los costes que la empresa debe incurrir por tener al empleado en plantilla (Seguridad social, depósitos por despido…) y no los costes de oficina (alquiler, electricidad, internet…).

8.3. Análisis económico

Para la evaluación de los costes finales del proyecto se tendrán en cuenta los costes en proporción al tiempo que fueron utilizados durante el éste, con respecto a la vida útil estimada de cada uno de ellos. Esto es debido a que, una vez terminado éste desarrollo, los equipos y programas utilizados se pueden seguir utilizando en el futuro.

Por motivos evidentes, sólo se detallarán los costes de aquellos elementos con precio mayor de 0€.

8.3.1 Costes estimados

En primer lugar, analizaremos los costes estimados si la planificación original se hubiese cumplido durante todo el desarrollo del proyecto.

Recurso	Precio	Vida útil estimada	Uso estimado	Coste para el proyecto
Máquina de desarrollo	1.100€	72 meses	3 meses	45,83€
Máquina de Backup/servidor SVN	630€	48 meses	3 meses	39,76€
Programador/analista	31.200€	12 meses	3 meses	7.800€
			Total:	**7.885,59€**

Tabla 31 – Costes estimados

Nota: El coste del Programador/Analista incluye tanto su salario bruto anual como los gastos adicionales de tenerlo en plantilla.

8.3.1 Costes reales

Debido a las desviaciones con respecto a la planificación original, los costes reales superan los costes estimados por un margen amplio ya que el tiempo total sido en torno al doble del planificado. Sin embargo, como se ha mencionado antes, aunque el desarrollo en sí casi ha doblado el tiempo planificado, se ha considerado que los costes de los periodos en los que sólo se han ejecutado pruebas de los agentes *Round Robin* y *UCT* (unos dos meses) serían más representativos de los reales si se estiman al 50% del coste mensual. Esto es debido a que, mientras se esperaba la finalización de los lotes de pruebas, los recursos destinados al proyecto se han empleado en tareas ajenas al mismo.

Recurso	Precio	Vida útil estimada	Uso real	Coste para el proyecto
Máquina de desarrollo	1.100€	72 meses	4,5 meses	68,75€
Máquina de Backup/servidor SVN	630€	48 meses	4,5 meses	59,06€
Programador/analista	31.200€	12 meses	4,5 meses	11.700€
			Total:	**11.827,81€**

Tabla 32 - Costes reales